THE UNIVERSITY OF CHICAGO

A PERFORMANCE STUDY OF THE GLOBUS TOOLKIT® AND GRID SERVICES VIA DIPERF, AN AUTOMATED DISTRIBUTED PERFORMANCE TESTING FRAMEWORK

A THESIS SUBMITTED TO

THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES

IN CANDIDACY FOR THE DEGREE OF

MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY

IOAN RAICU

ADVISOR

IAN FOSTER

COMMITTEE MEMBERS

IAN FOSTER, RICK STEVENS, JOHN REPPY

CHICAGO, ILLINOIS

MAY 2005

Abstract

The Globus Toolkit® (GT) is the "de facto standard" for grid computing. Measuring the performance of various components of the GT in a wide area network (WAN) as well as a local area network (LAN) is essential in understanding the performance that is to be expected from the GT in a realistic deployment in a distributed and heterogeneous environment. Furthermore, measuring the performance of grid services in a WAN is similarly important due to the complex interactions between network connectivity and service performance.

Performing distributed measurements is not a trivial task, due to difficulties 1) *accuracy* – synchronizing the time across an entire system that might have large communication latencies, 2) *flexibility* – in heterogeneity normally found in WAN environments and the need to access large number of resources, 3) *scalability* – the coordination of large amounts of resources, and 4) *performance* – the need to process large number of transactions per second. In attempting to address these four issues, we developed DiPerF, a DIstributed PERformance testing Framework, aimed at simplifying and automating service performance evaluation. DiPerF coordinates a pool of machines that test a single or distributed target service, collects and aggregates performance metrics from the client point of view, and generates performance statistics. The aggregate data collected provides information on service throughput, service response time, on service 'fairness' when serving multiple clients concurrently, and on the impact of network latency on service performance.

DiPerF is a modularized tool, with various components written in C/C++/perl; it has been tested over PlanetLab, Grid3, and the Computer Science Cluster at the University of Chicago. We have implemented several variations (ssh, TCP, UDP) on the communication between components in order to give DiPerF the most flexibility and scalability in a wide range of scenarios. We have investigated both the performance and scalability of DiPerF and found that DiPerF is able to handle up to 10,000+ clients and 100,000+ transactions per second; we also performed a validation study of DiPerF to ensure that the aggregate client view matched the tested service view and found that the two views usually matched within a few percent.

In profiling the performance of the GT, we investigated the performance of three main components: 1) job submission, 2) information services, and 3) a file transfer protocol. All these components of the Globus Toolkit are vital to the functionality, performance, and scalability of the grid. We specifically tested: 1) pre WS GRAM and WS GRAM included with GT 3.2 and 3.9.4, 2) the scalability and performance of the WS-MDS Index bundled with GT 3.9.5, and 3) the scalability and fairness of the GridFTP server included with the GT 3.9.5. We also investigated the performance of two grid services: 1) DI-GRUBER, a distributed usage SLA-based broker, a grid service based on the Globus Toolkit 3.2 and 3.9.5, and 2) instance creation and message passing performance in the Globus Toolkit 3.2.

Through the various test case studies we performed using DiPerF, we believe it has proved to be a valuable tool for scalability and performance evaluation studies, as well as for automated extraction of service performance characteristics. We conclude with future work that could benefit from the foundation established by DiPerF; we outline how we believe the performance models that can be extracted from DiPerF can be used to perform predictive scheduling in the Grid.

Table of Contents

Abstract 2
Table of Contents 3
List of Figures 6
List of Tables 9
1 Introduction 10
1.1 Motivation & Goals 10
1.2 Obstacles 11
1.3 Contributions 11
1.4 Thesis Outline 12
2 Related Work & Background Information 13
2.1 Grid Services and Grid Performance Studies Related Work 13
2.1.1 Grid Performance Studies 13
2.1.2 GRAM Performance Studies Related Work 14
2.1.3 MDS Performance Studies Related Work 14
2.1.4 GridFTP Related Work 14
2.1.5 DI-GRUBER Performance Studies Related Work 15
2.2 DiPerF Related Work – Distributed Performance Measurement Studies 15
2.3 Background Information on DiPerF components, test cases, and testbeds 17
2.3.1 Grid Computing 17
2.3.2 Testbeds 19
2.3.2.1 Grid3 19
2.3.2.2 PlanetLab 19
2.3.2.3 UChicago CS Cluster 20
2.3.2.4 DiPerF Cluster 20
2.3.2.5 Other Machines 21
2.3.3 Communication Protocols: a Layered Approach 22
2.3.3.1 TCP & UDP 23
2.3.3.2 SSH 23
2.3.4 Test Cases 24
2.3.4.1 Globus Toolkit 3.2 24
2.3.4.2 Globus Toolkit 3.9.5 24
2.3.4.3 GRAM 26
2.3.4.4 GridFTP 28
2.3.4.5 WS-MDS 28
2.3.4.6 GRUBER & DI-GRUBER 28
3 DiPerF Framework 30
3.1 Scalability Issues 33

3.2 Client Code Distribution ... 34
3.3 Clock Synchronization ... 34
3.4 Client Control and Performance Metric Aggregation ... 35
3.5 Performance Analyzer ... 36
4 DiPerF Scalability and Validation Study ... 39
4.1 Communication Performance ... 40
4.2 Time Server Performance via TELNET ... 41
4.3 Analyzer Performance ... 42
4.4 Overall DiPerF Performance and Scalability ... 43
4.4.1 Echo Performance via TCP ... 43
4.4.2 Echo Performance via UDP ... 44
4.4.3 Echo Performance via SSH ... 44
4.4.4 GridFTP: real service scaling to 1,000s of clients via SSH ... 45
4.5 Validation ... 46
5 Experimental Results ... 48
5.1 GRAM in GT3.2 & GT4 ... 48
5.1.1 GT3.2 pre-WS GRAM ... 48
5.1.2 GT3.2 WS GRAM ... 51
5.1.3 GT3.9.4 pre-WS GRAM and WS-GRAM Performance Results ... 54
5.1.3.1 WS-GRAM Results ... 54
5.1.3.2 pre-WS GRAM Results ... 56
5.1.3.3 GT3.9.4 GRAM Conclusions ... 59
5.1.4 GRAM Summary ... 59
5.2 GridFTP ... 59
5.2.1 GridFTP Scalability ... 60
5.2.2 GridFTP Fairness ... 65
5.2.3 GridFTP Conclusions ... 68
5.3 WS-MDS Index Scalability and Performance Results ... 69
5.3.1 WS-MDS Results ... 69
5.3.2 WS-MDS Conclusions ... 77
5.4 Grid Services ... 77
5.4.1 GT3.2 Instance Creation ... 77
5.4.2 DI-GRUBER: a distributed grid service ... 78
5.4.2.1 Experimental Settings & Performance Metrics ... 79
5.4.2.2 GT3-based Empirical Results ... 79
5.4.2.3 GT4-based Empirical Results ... 83
5.4.2.4 DI-GRUBER Conclusions ... 86
5.5 Other Work that has used DiPerF ... 86

6 Future Work 87
6.1 Related Work 88
6.2 Proposed Future Work 88
6.2.1 DiProfile: Perform software characterization 89
6.2.2 DiPredict: Automatic mapping of software requirements to raw resources & Service Performance Predictions 90
6.2.3 DiSched: Matchmaking from needed resources to available raw resources 91
6.3 Future Work Conclusion 92
7 Conclusion 93
8 Bibliography 95

List of Figures

Figure 1: Grid Computing Overview 18

Figure 2: PlanetLab Network Performance from 268 nodes to a node at UChicago as measured by IPERF on April 13th, 2005; each circle denotes a physical machine with the corresponding x-axis and y-axis values as its network characteristics, namely network latency and bandwidth. 19

Figure 3: PlanetLab Network Performance from 268 nodes to a node at UChicago as measured by IPERF on April 13th, 2005 shown with x and y axis in log scale. 20

Figure 4: OSI Reference Model 22

Figure 5: TCP/IP Reference Model; on the left the various levels are identified while on the right examples of functionality/protocol at each respective layer 22

Figure 6: Globus Toolkit System Overview [72] 24

Figure 7: GT4 Key Components [73] 25

Figure 8: WS GRAM Component Architecture Approach [76] 27

Figure 9: Partially ordered sequence of client activities around a WS GRAM job [76] 27

Figure 10: VO-Level Architecture 29

Figure 11: DiPerF demo showing the end of an experiment with 1000 clients accessing a TCP server 30

Figure 12: DiPerF framework overview 31

Figure 13: DiPerF framework overview deployment scenario 32

Figure 14: Aggregate view at the controller. Each tester synchronizes its clock with the time server every five minutes. The figure depicts an aggregate view of the controller of all concurrent testers. 36

Figure 15: Sample output from the automated graph generated by gnuplot (throughput – right hand axis, response time and load – left hand axis) 38

Figure 16: Sample output from the manual graph containing the same results (throughput – right hand axis) from Figure 15 plus two other metrics (response time and load) 38

Figure 17: DiPerF Components and Communication Overview 39

Figure 18: Summary of Communication Performance and Scalability: UDP, TCP, SSH RAW, and SSH 40

Figure 19: Summary of Communication Performance and Scalability 41

Figure 20: Time server performance with 1000 clients 42

Figure 21: Echo performance via TCP-based Communication Protocol 43

Figure 22: Echo performance via UDP Communication Protocol 44

Figure 23: Echo performance via SSH Communication Protocol 45

Figure 24: Scalability of DiPerF with 4000 GridFTP clients using SSH 45

Figure 25: DiPerF Validation against the Ganglia Monitor using GridFTP and 1300 clients in PlanetLab 46

Figure 26: DiPerF Validation against a TCP server using a TCP client and 1000 clients in PlanetLab 47

Figure 27: GT3.2 pre-WS GRAM: Service response time, throughput, and service load 49

Figure 28: GT3.2 pre-WS GRAM Service utilization per machine 50

Figure 29: GT3.2 pre-WS GRAM: Average aggregate load vs. number of requests completed per machine. The (x,y) coordinate of the bubble are the machine ID the average aggregate service load respectively; the size of each bubble is proportional to the number of jobs completed by each client... 51

Figure 30: GT3.2 WS GRAM: Response time, throughput, and load... 52

Figure 31: GT3.2 WS GRAM: Service utilization per machine 53

Figure 32: GT3.2 WS GRAM: Average aggregate load and the number of jobs completed per machine; see Figure 29 for the (x,y) coordinates.. 53

Figure 33: GT3.9.4 WS GRAM client (C implementation) and WS GRAM service (JAVA implementation); tunable parameters: utilized 69 concurrent nodes, starts a new node every 10 seconds, each node runs for 1000 seconds.. 55

Figure 34: GT3.9.4 WS GRAM client (C implementation) and WS GRAM service (JAVA implementation); tunable parameters: attempted to use 115 concurrent nodes, but after 72 concurrent clients started, the service became unresponsive; a new node was started every 25 seconds, and each node was scheduled to run for 3600 seconds.. 55

Figure 35: GT3.9.4 WS GRAM client (C implementation) with output enabled and WS GRAM service (JAVA implementation); tunable parameters: 8 concurrent clients, a new node was started every 10 seconds, and each node was scheduled to run for 1000 seconds 56

Figure 36: GT3.9.4 pre-WS GRAM client (C implementation) and pre-WS GRAM service (C implementation); tunable parameters: utilized 115 concurrent nodes, starts a new node every 25 seconds, each node runs for 3600 seconds.. 57

Figure 37: GRAM2 without output.. 57

Figure 38: GRAM2 with output.. 58

Figure 39: GRAM2 authentication... 58

Figure 40: GridFTP server performance with 100 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 100 concurrent clients, starts a new client every 30 seconds, each client runs for 7200 seconds; 243.4 GB of data transferred over 24,925 file transfers; left axis – load, response time; right axis - throughput.. 60

Figure 41: GridFTP server performance with 500 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 500 concurrent clients, starts a new client every 6 seconds, each client runs for 7200 seconds; 363.3 GB of data transferred over 37,201 file transfers; left axis – load, response time; right axis - throughput.. 61

Figure 42: GridFTP server performance with 1100 clients running on 100 physical nodes in PlanetLab and 30 physical nodes in the CS cluster at UofC; tunable parameters: utilized 1100 concurrent clients, starts a new client every 1 second, each client runs for 2400 seconds; 131.1 GB of data transferred over 13,425 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %... 62

Figure 43: GridFTP server performance with 1100 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 1100 concurrent clients, starts a new client every 6 seconds, each client runs for 14400 seconds; 916.58 GB of data transferred over 93,858 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %... 63

Figure 44: GridFTP server performance with 1300 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 1300 concurrent clients, starts a new client every 6 seconds, each client runs for 14400 seconds; 767 GB of data transferred over 78,541 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %... 64

Figure 45: GridFTP server performance with 1800 clients running on 100 physical nodes in PlanetLab and 30 physical nodes in the CS cluster at UofC; tunable parameters: utilized 1800 concurrent clients, starts a new client

every 1 second, each client runs for 2400 seconds; 150.7 GB of data transferred over 15,428 file transfers; left axis – load, response time, memory; right axis – throughput, CPU % 65

Figure 46: GridFTP resource usage per machine using 10MB files 66

Figure 47: GridFTP resource usage per client using 10MB files 66

Figure 48: GridFTP resource usage per machine using 1MB files 67

Figure 49: GridFTP resource usage per client using 1MB files 68

Figure 50: WS-MDS Index LAN Tests with no security including 4 clients running on 4 physical nodes at UChicago in a LAN connected via 1 Gb/s links; tunable parameters: utilized 4 concurrent clients, with each client starting every 15 seconds; left axis – load, response time; right axis – throughput 70

Figure 51: WS-MDS Index LAN Tests with no security including 100 clients running on 4 physical nodes at UChicago in a LAN connected via 1 Gb/s links; tunable parameters: utilized 100 concurrent clients, with each client starting every 15 seconds; left axis – load, response time; right axis – throughput 71

Figure 52: WS-MDS Index WAN Tests with no security including 128 clients running on 128 physical nodes in PlanetLab in a WAN connected via 10 Mb/s links; tunable parameters: utilized 128 concurrent clients, with each client starting every 3 seconds; left axis – load, response time; right axis – throughput 72

Figure 53: WS-MDS Index WAN Tests with no security including 288 clients running on 288 physical nodes in PlanetLab in a WAN connected via 10 Mb/s links; tunable parameters: utilized 288 concurrent clients, with each client starting every 2 seconds; left axis – load, response time; right axis – throughput 73

Figure 54: WS-MDS Index LAN+WAN Tests with no security including 100 clients running on 3 physical nodes at UChicago (LAN) and 97 physical nodes in PlanetLab (WAN); tunable parameters: utilized 100 concurrent clients, with each client starting every 2 seconds; left axis – load, response time; right axis – throughput 74

Figure 55: WS-MDS Index LAN+WAN Tests with no security including 100 clients running on 3 physical nodes at UChicago (LAN) and 97 physical nodes in PlanetLab (WAN); tunable parameters: utilized 100 concurrent clients, with each client starting every 2 seconds; left axis – load, LAN resource utilization %; right axis – throughput 75

Figure 56: WS-MDS Index LAN Tests with security including 4 clients running on 4 physical nodes at UChicago in a LAN connected via 1 Gb/s links; tunable parameters: utilized 4 concurrent clients, with each client starting every 15 seconds; left axis – load, response time; right axis – throughput 76

Figure 57: WS-MDS Index WAN Tests with security including 128 clients running on 128 physical nodes in PlanetLab in a WAN connected via 10 Mb/s links; tunable parameters: utilized 128 concurrent clients, with each client starting every 3 seconds; left axis – load, response time; right axis – throughput 76

Figure 58: GT3.2 Service Instance Creation: Response time, Throughput, and Load 78

Figure 59: Centralized GT3-based Scheduling Service 80

Figure 60: GT3-based Distributed Scheduling Service with 3 Decision Points 81

Figure 61: GT3-based Distributed Scheduling Service with 10 Decision Points 82

Figure 62: GT4-based Centralized Scheduling Service 83

Figure 63: GT4-based Distributed Scheduling Service with 3 Decision Points 84

Figure 64: GT4-based Distributed Scheduling Service with 10 Decision Points 85

Figure 65: Proposed system overview 89

Figure 66: Simple example showing that co-scheduling software that has different requirements could be run in parallel without loss of performance; note that in a serial fashion, it would take 299 seconds to complete all 3 jobs, while if we run the first 2 jobs concurrently, it would only take 200 seconds 91

List of Tables

Table 1: DiPerF cluster at UChicago hosts hardware and OS details 21

Table 2: Host specifications of machine used in GridFTP tests located at ISI... 21

Table 3: Analyzer Performance 42

Table 4: GRAM performance summary covering pre-WS GRAM and WS-GRAM under both GT3 and GT4 .. 59

Table 5: GridFTP WAN performance summary ranging from 100 clients up to 1800 clients 69

Table 6: WS-MDS summary of experiments in both LAN and WAN with both security enabled and disabled; for the “load at service saturation, the * indicates that the service was not saturated with the number of concurrent clients that were used 77

Table 7: GT3-based distributed scheduling service summary with 1, 3, and 10 decision points peak response time in seconds 82

Table 8: GT3-based distributed scheduling service summary with 1, 3, and 10 decision points peak throughput (queries per second) and the load at service saturation (* indicates that the service was not saturated with the number of concurrent clients that were used) 82

Table 9: GT4-based distributed scheduling service summary with 1, 3, and 10 decision points peak response time in seconds 85

Table 10: GT4-based distributed scheduling service summary with 1, 3, and 10 decision points peak throughput (queries per second) and the load at service saturation (* indicates that the service was not saturated with the number of concurrent clients that were used) 85

1 Introduction

The Globus Toolkit is the "de facto standard" for grid computing as it has been called by numerous agencies and world recognized news sources such as the New York Times. Measuring the performance of the Globus Toolkit is important to ensure that the Grid will continue to grow and scale as the user base and infrastructure expands. Furthermore, the testing of the toolkit and grid services is difficult due to the distributed and heterogeneous environment Grids are usually found in. Performing distributed measurements is not a trivial task, due to difficulties 1) *accuracy* – synchronizing the time across an entire system that might have large communication latencies, 2) *flexibility* – in heterogeneity normally found in WAN environments and the need to access large number of resources, 3) *scalability* – the coordination of large amounts of resources, and 4) *performance* – the need to process a large number of transactions per second. In attempting to address these four issues, we developed DiPerF, a DIstributed PERformance testing Framework, aimed at simplifying and automating service performance evaluation. DiPerF coordinates a distributed pool of machines that run clients of a target service, collects and aggregates performance metrics, and generates performance statistics. The data collected provides information on a particular service's maximum throughput, on service 'fairness' when multiple clients access the service concurrently, and on the impact of network latency on service performance from both client and service viewpoint. Using this data, it may be possible to build empirical performance estimators that link observed service performance (throughput, response time) to offered load. These estimates can be then used as input by a resource scheduler to increase resource utilization while maintaining desired quality of service levels. All steps involved in this process are automated, including dynamic deployment of a service and its clients, testing, data collection, and the data analysis.

DiPerF is a modularized tool, with various components written in C/C++/perl; it has been tested over PlanetLab, Grid3, and the Computer Science Cluster at the University of Chicago. We have implemented several variations (ssh, TCP, UDP) of the communication between components in order to give DiPerF the most flexibility and scalability in a wide range of scenarios. We have investigated both the performance and scalability of DiPerF and found that DiPerF is able to handle up to 10,000+ clients and 100,000+ transactions per second; we also performed a validation study of DiPerF to ensure that the aggregate client view matched the tested service view and found that the two views usually matched within a few percent.

In profiling the performance of the Globus Toolkit, we investigated the performance of three main components: 1) job submission, 2) information services, and 3) a file transfer protocol. All these components of the Globus Toolkit are vital to the functionality, performance, and scalability of the grid. We specifically tested: 1) pre WS GRAM and WS GRAM included [1, 2, 3, 4, 5] with Globus Toolkit 3.2 and 3.9.4, 2) the scalability and performance of the WS-MDS Index bundled with Globus Toolkit 3.9.5, and 3) the scalability and fairness of the GridFTP server included with the Globus Toolkit 3.9.5. We also investigated the performance of two grid services: 1) DI-GRUBER, a distributed usage SLA-based broker, a grid service based on the Globus Toolkit 3.2 and 3.9.5, and 2) instance creation and message passing performance in the Globus Toolkit 3.2. Through the various test case studies we performed using DiPerF, we believe it has proved to be a valuable tool for scalability and performance evaluation studies, as well as for automated extraction of service performance characteristics.

1.1 Motivation & Goals

Multiple threads motivated me to measure the performance of the Globus Toolkit, which in turn motivated the building of DiPerF. The Globus Toolkit is the "de facto standard" for grid computing. Measuring the performance of the various components of the Globus Toolkit in a WAN as well as a LAN is essential in understanding the performance that is to be expected from the Globus Toolkit in a realistic deployment in a distributed and heterogeneous environment. Furthermore, measuring the performance of grid services in a WAN is similarly important due to the complex interactions between network connectivity and service performance.

Although performance testing is an 'everyday' task, testing harnesses are often built from scratch for a particular service. DiPerF can be used to test the scalability and performance limits of a service: that is, find the maximum offered load supported by the service while still serving requests with an acceptable quality of service. Actual

service performance experienced by heterogeneous and geographically distributed clients with different levels of connectivity cannot be easily gauged based on controlled LAN-based tests. Therefore significant effort is sometimes required in deploying the testing platform itself. With a wide-area, heterogeneous deployment provided by the PlanetLab [6, 7] and Grid3 [8] testbed, DiPerF can provide accurate estimation of the service performance as experienced by such clients.

1.2 Obstacles

Automated performance evaluation and result aggregation across a distributed test-bed is complicated by multiple factors. In building DiPerF, we encountered 4 main obstacles:

- accuracy – time synchronization
- flexibility: heterogeneity in WAN environments & accessing of many resources
- scalability – coordination of many resources
- performance – processing large number of transactions per second

The accuracy of the performance metrics collected is heavily dependent on the accuracy of the timing mechanisms used and on accurate clock synchronization among the participating machines. DiPerF synchronizes the time between client nodes with a synchronization error smaller than 100ms on average. The reliability of presented results is important, especially in wide-area environments: we detect client failures during the test that could impact on reported result accuracy.

The heterogeneity normally found in WAN environments pose a challenging problem for any large scale testing due to different remote access methods, different administrative domains, different hardware architectures, and different operating systems and host environments. We have shown DiPerF to be flexible by implementing the support for three testbeds: Grid3, PlanetLab, and the University of Chicago CS cluster. Grid3 offers a testbed in which DiPerF uses the Globus Toolkit as the main method of deploying clients and retrieving performance metrics. PlanetLab offers a unique environment where there is a uniform remote access method, the client code gets deployed via rsync, and the communication is implemented via ssh, TCP, or UDP. The UChicago CS cluster is similar to that of PlanetLab, however it has the advantage of having a network file system (NFS) which make the deployment of clients almost trivial; on the other hand, the communication occurs in exactly the same manner as it does in PlanetLab.

The scalability of the framework itself is important; otherwise DiPerF will not be able to saturate a target service. We insure scalability by only loosely coupling the participating components, and by having multiple implementations of communication protocols between components. DiPerF has been designed and implemented to be scalable to 10,000+ clients that could generate 100,000+ transactions per second.

DiPerF has been measured to process up to 200,000 transactions per second via TCP and up to 15,000 transactions per second via SSH. The performance of DiPerF has been carefully tuned to use the most lightweight protocols and tools in order to achieve its goals. For example, the communication protocol built on TCP uses a single process and the select() function to multiplex the 1,000s of concurrent connections. The structures that are used to store and transfer the performance metrics have been optimized for space and efficiency. Furthermore, each TCP connection’s buffering is kept to a minimum in order to lower the memory footprint of DiPerF and to ensure the desired scalability; the drawback of the small memory footprint per connection is the limited connection bandwidth that could be achieved per connection, but with 1,000s of concurrent connections, this would hardly be an issue.

In summary, DiPerF has been designed from the ground up with scalability, performance, flexibility, and accuracy as its target goal, and based on the results in this thesis, we believe this target goal has been achieved.

1.3 Contributions

The contributions of this thesis are two fold: 1) a detailed empirical performance analysis of various components of the Globus Toolkit along with a few grid services, and 2) DiPerF, a tool that makes automated distributed performance testing easy.

Through our tests performed on GRAM, WS-MDS, and GridFTP, we have been able to quantify the performance gain or loss between various different versions or implementations, and have normally found the upper limit on both scalability and performance on these services. We have also been able to show the performance of these components in a WAN, a task that would have been very tedious and time consuming without a tool such as DiPerF. By pushing the Globus Toolkit to the limit in both performance and scalability, we was able to give the users a rough overview of the performance they are to expect so they can do better resource planning. The developers also gained feedback on the behavior of the various components under heavy stress and allowed them to concentrate on improving the parts that needed the most improvements. We were also able to quantify the performance and scalability of DI-GRUBER, a distributed grid service built on top of the Globus Toolkit 3.2 and the Globus Toolkit 3.9.5.

The second main contribution is DiPerF itself, which provides a tool that allows large scale testing of grid services, web services, and network services to be done in both LAN and WAN environments. DiPerF has been automated to the extent that once configured, the framework will automatically do the following steps:

- check what machines or resources are available for testing
- deploy the client code on the available machines
- perform time synchronization
- run the client code in a controlled and predetermined fashion
- collect performance metrics from all the clients
- stop and clean up the client code from the remote resources
- aggregate the performance metrics at a central location
- summarize the results
- generates graphs depicting the aggregate performance of the clients and tested service

In summary, DiPerF offers an easy solution to large scale distributed performance measurements.

1.4 Thesis Outline

This thesis is organized in several chapters as follows. The rest of Chapter 1 covers the motivation, goals, obstacles, and contributions of this work. Chapter 2 covers an extensive related work and background information. We discuss related work for grid performance studies in general, GRAM, MDS, GridFTP, and DI-GRUBER performance studies. We then discuss other work similar to DiPerF, specifically distributed performance measurement frameworks, tools, and studies. An extensive background information section then follows discussing everything from the definition of Grid Computing, to the description of the various testbeds (Grid3, PlanetLab, UChicago CS Cluster, DiPerF Cluster) used to run all the experiments in this thesis, to a brief overview of communication protocols (TCP, UDP, SSH), to a description of the various components tested (Globus Toolkit 3.2, GT3.9.5, GRAM, GridFTP, WS-MDS, and DI-GRUBER). Chapter 3 discusses the DiPerF framework including scalability issues, client code distribution, clock synchronization, client control, metric aggregation, and the performance analyzer. Chapter 4 covers the scalability, performance, and validation study of DiPerF. Chapter 5 discusses all the experimental results from testing the Globs Toolkit and grid services; Chapter 6 introduces some new ideas that could benefit from the foundation established by DiPerF; we outline how we believe the performance models that can be extracted from DiPerF can be used to perform predictive scheduling in the Grid. We conclude with Chapter 7 summarizing the entire thesis and its contributions.

2 Related Work & Background Information

This section covers the related work to DiPerF including similar studies performed on grid services and various grid components. It also addresses basic and introductory information on various topics in order to make this thesis self contained.

2.1 Grid Services and Grid Performance Studies Related Work

We first cover related work for grid performance studies in general, studies on various components such as GridFTP, MDS, and GRAM, and finally the related work to the grid services we tested.

2.1.1 Grid Performance Studies

NetLogger [9] targets instrumentation of Grid middleware and applications, and attempts to control and adapt the amount of instrumentation data produced in order not to generate too much monitoring data. NetLogger is focusing on monitoring, and requires code modification in the clients; furthermore, it does not address automated client distribution or automatic data analysis.

GridBench [10] provides benchmarks for characterizing Grid resources and a framework for running these benchmarks and for collecting, archiving, and publishing results. While DiPerF focuses on performance exploration for entire services, GridBench uses synthetic benchmarks and aims to test specific functionalities of a Grid node. However, the results of these benchmarks alone are probably not enough to infer the performance of a particular service.

The development team of the Globus Toolkit have done extensive testing [11, 12] of the Globus Toolkit in LAN environments. Some of the tests they performed are even more involved and complex than what we have tested in this work, but the downside of these results is the artificial environment that is created in a LAN setup with multiple clients running on few machines. The results we obtained with 100s of machines distributed all over the world are much more likely to depict the realistic performance of the various Globus Toolkit components.

Grid applications can combine the use of compute, storage, network, and other resources. These resources are often geographically distributed, adding to application complexity and thus the difficulty of understanding application performance. GridMapper [13] is a tool for monitoring and visualizing the behavior of such distributed systems. GridMapper builds on basic mechanisms for registering, discovering, and accessing performance information sources, as well as for mapping from domain names to physical locations. The visualization system itself then supports the automatic layout of distributed sets of such sources and animation of their activities.

In grid computing environments, network bandwidth discovery and allocation is a serious issue. Before their applications are running, grid users will need to choose hosts based on available bandwidth. Running applications may need to adapt to a changing set of hosts. Hence, a tool is needed for monitoring network performance that is integral to the grid environment. To address this need, Gloperf [14] was developed as part of the Globus grid computing toolkit. Gloperf is designed for ease of deployment and makes simple, end-to-end TCP measurements requiring no special host permissions. Scalability is addressed by a hierarchy of measurements based on group membership and by limiting overhead to a small, acceptable, fixed percentage of the available bandwidth.

The Network Weather Service (NWS) [15] is a distributed monitoring and forecasting system. A distributed set of performance sensors feed forecasting modules. There are important differences to DiPerF. First, NWS does not attempt to control the offered load on the target service but merely to monitor it. Second, the performance testing framework deployed by DiPerF is built on the fly, and removed as soon as the test ends; while NWS sensors aim to monitor network performance over long periods of time.

2.1.2 GRAM Performance Studies Related Work

The Globus Toolkit's 3.2 job submission service test suite [16] uses multiple threads on a single node to submit an entire workload to the server. However, this approach does not gauge the impact of a wide-area environment, and does not scale well when clients use many resources, which means that the service will be relatively hard to saturate. The Globus Toolkit 3.9.4 job submission was also partially tested by the same group [12], but the tests are incomplete, and do not cover nearly the level of detail that the tests presented in this work.

2.1.3 MDS Performance Studies Related Work

Zhang et al. [17] compared the performance of three resource selection and monitoring services: the Globus Monitoring and Discovery Service (MDS), the European Data Grid Relational Grid Monitoring Architecture (R-GMA) and Hawkeye. Their experiment uses two sets of machines (one running the service itself and one running clients) in a LAN environment. The setup is manual and each client node simulates 10 users accessing the service. This is exactly the scenario where DiPerF would have proved its usefulness: it would have freed the authors from deploying clients, coordinating them, and collecting performance results, and would allow focusing on optimally configuring and deploying the services to test, and on interpreting performance results. In a later study, Zhang et al. [18] investigated the performance of MDS using NetLogger and improved the testbed by having a cluster of 7 machines to handle the server side services and a cluster of 20 machines to handle the client side testbed; unfortunately, their study only addressed the performance of MDS in a LAN environment. On the other hand, our study of MDS was on a larger scale, utilizing over 100 machines in a WAN environment. We also tested the latest release of MDS (WS-MDS), based on the Globus Toolkit 3.9.5, which is the latest MDS implementation based on web services (WS).

Aloisio et al. [19] studied the capabilities and limitations of the Globus Toolkit's Monitoring and Discovery Service; however, their experiments were limited to simple tests on a Grid Index Information Service (GIIS) only.

2.1.4 GridFTP Related Work

The Globus Toolkit development team have performed their own performance tests [12] on the GridFTP server; some additional results (including some results obtained via DiPerF) can also be found in the Zebra work by Allcock et al. [20]. Our results obtained with DiPerF were complimentary to the already rich set of results existing in the Zebra [20] paper.

Kola et al [21] used the NetLogger tool to measure the performance of the GridFTP 2.4.3. GridFTP has a variety of tuning parameters; some parameters improve single client performance at the expense of increased server load, thereby limiting the number of served clients; on the other hand, other parameters increase the flexibility or security of the system at some expense. The objective of this study was to make such trade-offs clear and enable easy full system optimization. Beside the fact that the results reflect an old version of GridFTP, they also used a much smaller testbed of only a few machines, and examined the performance of GridFTP with less than 60 clients.

Baer et al. [22] performed a comparison between GridFTP and NFS using several I/O benchmarks. Their testbed was again relatively small and the tests were performed in a cluster over a LAN.

Finally, Liu et al. [23] implemented a tool called MicroGrid which allows researchers to emulate Grid resources and applications directly. The MicroGrid tools include a scalable network emulatorMaSSF whose design can support emulation thousands of grid resources and application elements. One of their test cases to demonstrate the MicroGrid tool was a detailed evaluation of GridFTP on the TeraGrid. Their studies were aimed at exploiting the packet-level simulation capabilities of MaSSF to expose detailed dynamic communication behavior. The limitation of their work is mainly in the fact that their results are obtained via simulations, especially when large scale testing (similarly sized to their simulations) is achievable via the DiPerF tool.

As a summary, our experiments were mainly geared towards the scalability and fairness of the GridFTP server, using 1000+ clients to concurrently access the FTP server over a WAN testbed. None of the related work addressed the scalability or fairness issues we addressed, but rather most of the related work addressed only the performance implications of the GridFTP server.

2.1.5 DI-GRUBER Performance Studies Related Work

Dumitrescu et al. [24, 25, 26, 27] has developed, implemented, and tested DI-GRUBER. The first paper [24] introduced the GRUBER framework and presented some preliminary performance numbers. The second paper [25, 26] made a scalability and performance evaluation of an improved version of GRUBER named DI-GRUBER which was implemented as a distributed service; some results from this thesis was also included in this paper. Finally, the last paper [27] addressed the performance of running workloads over a Grid using DI-GRUBER. In summary, these three papers cover the implementation and the performance of DI-GRUBER in a wide range of configurations.

2.2 DiPerF Related Work – Distributed Performance Measurement Studies

There are a number of wide area measurement projects that mostly focus on general network measurement, monitoring and analysis, but not on services or grids in general. The next few paragraphs will attempt to summary each work and highlight any limitations or differences when compared to DiPerF.

The National Internet Measurement Infrastructure (NIMI) project [28, 29] is the next generation of Paxson's NPD study [30]. The project's goal is to develop and deploy a secure management platform for facilitating general types of coordinated measurement in the Internet. The software has been distributed on 51 systems across the Internet at present. NIMI is not focused on a single measurement objective and is currently used in a number of different studies.

AT&T Research has been making detailed measurements in AT&T Worldnet for a number of years. Beginning with the development of their PacketScope passive measurement system [31], they have been able to extract large volumes of detailed packet traces and have developed methods for reconstructing Web browsing sessions from those traces [32].

The IETF Internet Protocol Performance Metrics (IPPM) [33, 34] working group's mission is to "develop a set of standard metrics that can be applied to the quality, performance, and reliability of Internet data delivery services." The project which has grown out of the IPPM work is called Surveyor [35]. This project, facilitated by Advanced Network Systems, is focused on taking one-way measurements of packet delay and loss between 61 systems deployed world wide. The most important aspect of this system from our perspective is that it has been constructed with Global Positioning Satellite (GPS) clocks which enable very precise synchronization between their systems.

Network Analysis Infrastructure (NAI) projects at the National Laboratory for Applied Network Research (NLANR) [36] are focused in three areas. The first is their Active Measurement Program (AMP). This is an infrastructure consisting of a confederation of approximately 100 universities principally in North America that participate in on-going round trip time, loss and topology measurements [37]. The second is their Passive Measurement and Analysis (PMA) program [38]. This is an infrastructure consisting of a confederation of 17 sites that participate in on-going passive measurements of throughput and packet flows. The third project is NLANR's Squid cache hierarchy [39]. This is a series of 8 caches distributed in the Internet which collect logs and publish them on a daily basis.

Keynote Systems, Inc. [40] has an infrastructure of over 1,000 measurement systems deployed world wide which they use to measure the performance of Web sites and, to a lesser extent, Internet Service Providers. Their principal service is to provide measurements of Web server download time to their customers, who are either content providers or content delivery companies. While Keynote provides a useful service for their customers, they provide only DNS response time and document download time. They are not able to differentiate between network and server delay in their system. Keynote also publishes a "performance index" for both consumer and business sites. This index attempts to assign a single "health" index value to the 40 most heavily used Web sites in each category.

The Collaborative Advanced Interagency Research Network (CAIRN) is a national passive measurement test bed consisting of 31 installations in government, academic and industrial sites nationwide [41]. Their primary focus has been to provide a set of programmable routers in the network for general research. However, in the future, they intend to broaden their scope to enable co-location of host measurement systems at their sites.

Cooperative Association for Internet Data Analysis (CAIDA) [42, 43] manages a variety of measurement and data collection efforts. At the structural level of the Internet, there are also several measurement studies of Internet topology. The largest of these is the Skitter project being conducted at CAIDA [44]. This project consists of 17 source sites which send out traceroute [45] probes to approximately 54,000 destination sites world wide. The skitter tool is used to visualize topology and performance attributes of a large cross- section of the Internet by probing the path from a few sources to many thousands of destinations spread throughout the IPv4 address space. CAIDA also developed CoralReef, a comprehensive software suite providing a set of drivers, libraries, utilities, and analysis software for passive network measurement of workload characteristics. These reports characterize workload on a high-speed link between UCSD and the commodity Internet. [46]

NETI@home [47] is an open-source software package that collects network performance statistics from end-systems. These statistics are then sent to a server at the Georgia Institute of Technology, where they are collected and made publicly available. This tool gives researchers much needed data on the end-to-end performance of the Internet, as measured by end-users. The basic approach is to sniff packets sent from and received by the host and infer performance metrics based on these observed packets. NETI@home is designed to be an unobtrusive software system that runs quietly in the background with little or no intervention by the user, and using few resources.

The CoSMoS system [48] is a performance monitoring tool for distributed programs executing on loosely coupled systems such as workstation clusters. The monitor is capable of capturing measurement data at the application level, the operating system level, and the hardware level. In order to compensate for the lack of a globally synchronous clock on loosely coupled systems, the monitor provides a mechanism for synchronizing the measurement data from different machines. Visualization components graphically present the measurement data gathered from the various sources in an integrated and synchronized fashion. The monitoring system is designed to encourage a well-directed technique of tracking down performance bottlenecks in multiple iterations of measurement, analysis, and refinement.

Remos [49] provides resource information to distributed applications. Its design goals of scalability, flexibility, and portability are achieved through an architecture that allows components to be positioned across the network, each collecting information about its local network. To collect information from different types of networks, Remos provides several Collectors that use different technologies, including SNMP and benchmarking. By matching the Collector to the particular network environment and by providing an architecture for distributing the output of these collectors across all querying environments, Remos collects appropriately detailed information at each site and distributes this information where needed in a scalable manner.

SLAC/DOE/ESnet, High Energy and Nuclear Physics uses pingER tools on 31 monitoring sites to monitor network performance for over 3000 links in 72 countries. Monitoring includes many major national networks (including ESNet, vBNS, Internet2-Abilene, CALREN2, NTON, and MREN) as well as networks in South America, Canada, Europe, the former Soviet Union, New Zealand, and Africa. [50]

The Measurement and Analysis of Wide-area Internet (MAWI) Working Group studies performance of networks and networking protocols in Japanese wide-area networks. Sponsored by the Widely Integrated Distributed Environment (WIDE) project, MAWI is a joint effort of Japanese network research and academic institutions with corporate sponsorship. [51]

The PPNCG (Particle Physics Network Coordinating Group) runs network monitoring processes on machines at several sites throughout Europe. Its goal is to gather end-to-end performance information for links of specific interest to particle physics researchers, and use the information to highlight problems and help the PPNCG to make recommendations to the appropriate bodies to optimize the networking available to the UK particle physics community. This project uses Traceping Route Monitoring Statistics. [52]

Canadian national research facility uses Perl scripts to trace paths toward nodes of interest to TRIUMF. Packet loss summaries and graphs are generated daily from pins made at 10 minute intervals. Traceroute data is gathered four times daily. Network visualization maps are generated from the traceroute data. [53]

The WAND project aims to build models of internet traffic for statistical analysis and for the construction of simulation models. The project builds its own measurement hardware and collects and archives significant network traces. These are used internally and are also made available to the Internet research community. Traces

are accurately time stamped and synchronized to GPS. Many traces are 24 hours long, some are up to a week long, and there are plans to provide even longer traces in the future. The WAND project is based at the University of Waikato in New Zealand with strong collaboration from the University of Auckland. [54]

Measures "traffic index", response time, and packet loss by pinging many routers along "major paths" on the Internet. (traffic index - a score from 0 to 100 where 0 is "slow" and 100 is "fast", determined by comparing the current response to a ping echo request with all previous responses from the same router for past 7 days.) [55]

The MIDS Internet Average is a high-level summary of Internet performance measured from hosts all around the world. It provides one baseline against which more specialized Internet performance data might be compared, serving a similar role as the Dow Jones Industrial Average does in the financial world. The MIDS IWR presents ongoing animated scans of macroscopic conditions across the Internet. IWR displays geographical maps that show ping-based RTT latency from MIDS offices in Austin, Texas to thousands of Internet domains worldwide. Data is currently updated every four hours, six times a day, seven days a week. MIDS monitors thousands of sites worldwide every 15 minutes to map the data flow of the Internet. Statistical analyses of this data to determine network performance form the basis for their Matrix Internet Quality (MIQ) products. Only a small fraction of the information capable of being provided by MIQ is used in this public ratings page. [56, 57, 58]

NetSizer provides daily and monthly statistics on the size and growth of the Internet, including the number of computer hosts in the public Internet by top level domain and by second level domain. NetSizer also provides statistics on Internet penetration by country measured in terms of the number of hosts and Internet users. A separate report on the overall Internet quality presents results of active measurements to a set of about 100 URLs considered by Telcordia to be representative of the entire Internet. Telcordia's methodology is described here, and Sam Weerahandi is the contact for any further questions or information. [59]

Finally, Web server performance has been a topic of much research. The Wide Area Web Measurement (WAWM) Project for example designs an infrastructure distributed across the Internet allowing simultaneous measurement of web client performance, network performance, and web server performance [60]. Banga et al. [61] measure the capacity of web servers under realistic loads. Both systems could have benefited from a generic framework such as DiPerF.

Each of these projects attempts to measure or analyze some aspect of Internet/network performance. Most of the projects listed above focus on general network measurement, monitoring and analysis, rather than service/application performance, and the few projects that do concentrate on service level performance, often have been deployed in a very small testbed and have not taken into consideration a wide area environment; furthermore, some of the works also require client side source code modifications.

2.3 Background Information on DiPerF components, test cases, and testbeds

This section contains a brief overview of various key concepts, testbeds, and software packages used in this work in order to make this thesis self contained. It covers 1) the definition of Grid Computing, 2) the description of four different testbeds (Grid3, PlanetLab, UChicago CS cluster, and the DiPerF cluster), 3) communication protocols (ssh, TCP, UDP), and 4) the background information on the various test cases (Globus Toolkit 3.2, 3.9.5, GRAM, GridFTP, WS-MDS, and GRUBER).

2.3.1 Grid Computing

The term “the Grid” was coined in the mid 1990s to denote a proposed distributed computing infrastructure for advanced science and engineering [62]. Foster et al. describes the definition of Grid Computing through a series of excerpts taken from the famous paper from 2001 “The Anatomy of the Grid: Enabling Scalable Virtual Organizations” [3].

> *“’Grid’ computing has emerged as an important new field, distinguished from conventional distributed computing by its focus on large-scale resource sharing, innovative applications, and, in some cases, high-performance orientation.”*
>
> *“... the ‘Grid problem,’ which we define as flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources—what we refer to as virtual*

organizations. In such settings, we encounter unique authentication, authorization, resource access, resource discovery, and other challenges. It is this class of problem that is addressed by Grid technologies."

"The real and specific problem that underlies the Grid concept is coordinated resource sharing and problem solving in dynamic, multi-institutional virtual organizations. The sharing that we are concerned with is not primarily file exchange but rather direct access to computers, software, data, and other resources, as is required by a range of collaborative problem-solving and resource brokering strategies emerging in industry, science, and engineering. This sharing is, necessarily, highly controlled, with resource providers and consumers defining clearly and carefully just what is shared, who is allowed to share, and the conditions under which sharing occurs. A set of individuals and/or institutions defined by such sharing rules form what we call a virtual organization (VO)."

As can be inferred by the definition of Grid Computing given by Foster et al. [3], Grid Computing is about large scale resource sharing, innovative applications, and high performance computing. It is meant to offer flexible, secure, coordinated resource sharing among dynamic collections of individuals, institutions, and resources, namely virtual organizations. Figure 1 below gives a glimpse into the complex nature of Grid Computing; it depicts virtual organizations (blue clouds) interconnected at both the physical layer (via networks) and at the abstract layer (via service layer agreements). Each virtual organization has resources (i.e. direct access to computers, software, data) and users. Based on the agreements among the various VOs, users can locate and share resources that are part of different virtual organizations and could be physically located anywhere in the world.

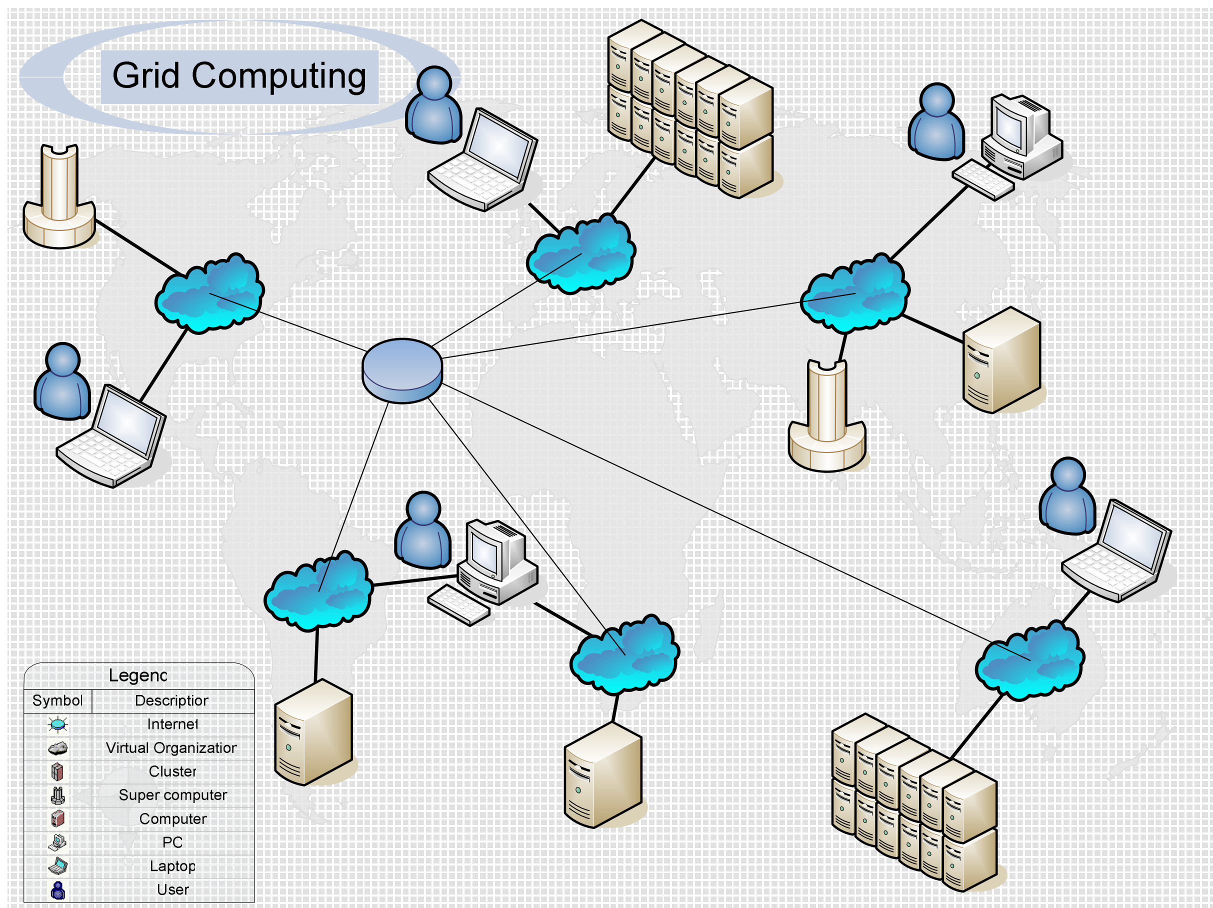


Figure 1: Grid Computing Overview

2.3.2 Testbeds

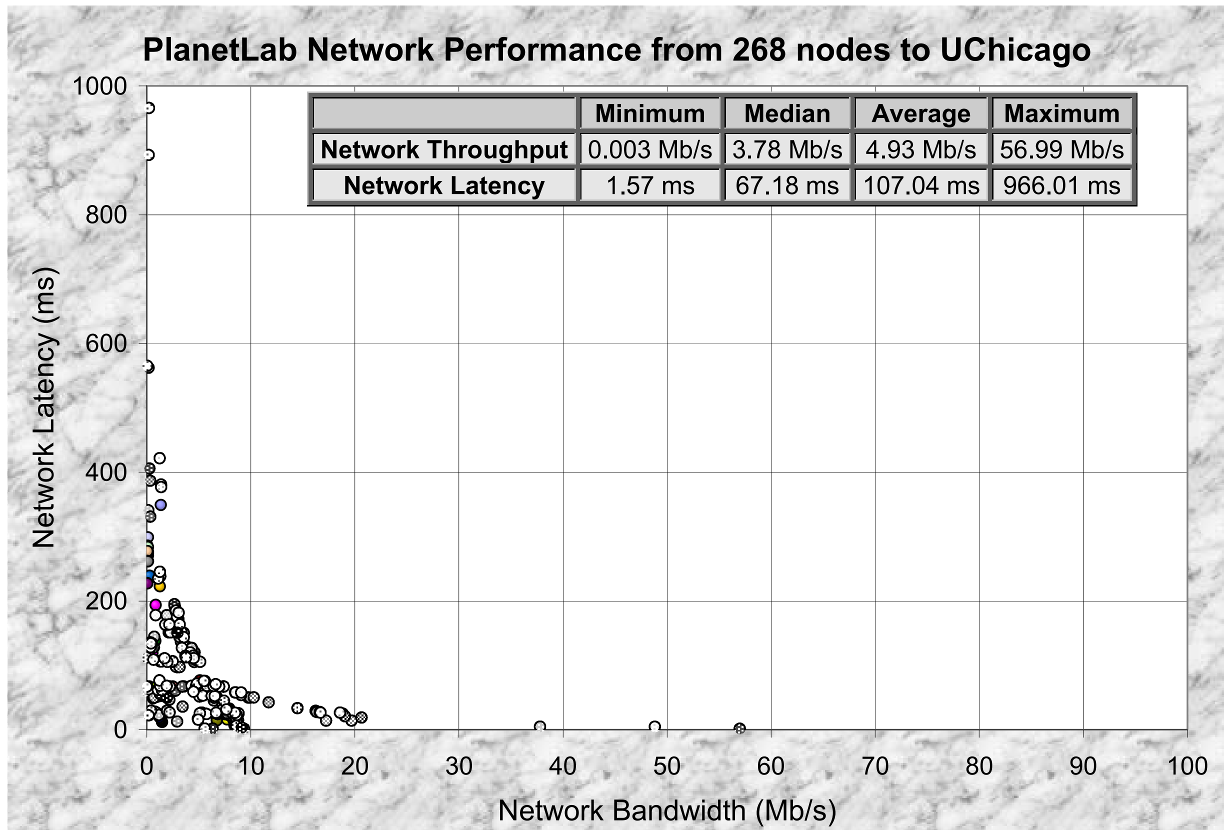
This section covers the four testbeds (Grid3, PlanetLab, the University of Chicago CS cluster, and the DiPerF cluster) that we used in this work. For each set of experiments in this work, we outline what testbed we used; this is an important section since the results of certain tests might vary with the particular testbed.

2.3.2.1 Grid3

The Grid3 collaboration has deployed an international Data Grid with dozens of sites and thousands of processors. The facility is operated jointly by the U.S. Grid projects iVDGL, GriPhyN and PPDG, and the U.S. participants in the LHC experiments ATLAS and CMS. Participation is combined across more than 25 sites which collectively provide more than 4000 CPUs and over 2 TB of aggregate memory. The resources are used by 7 different scientific applications, including 3 high energy physics simulations and 4 data analyses in high energy physics, bio-chemistry, astrophysics and astronomy. More than 100 individuals are currently registered with access to the Grid, with a peak throughput of 500-900 jobs running concurrently. [63]

2.3.2.2 PlanetLab

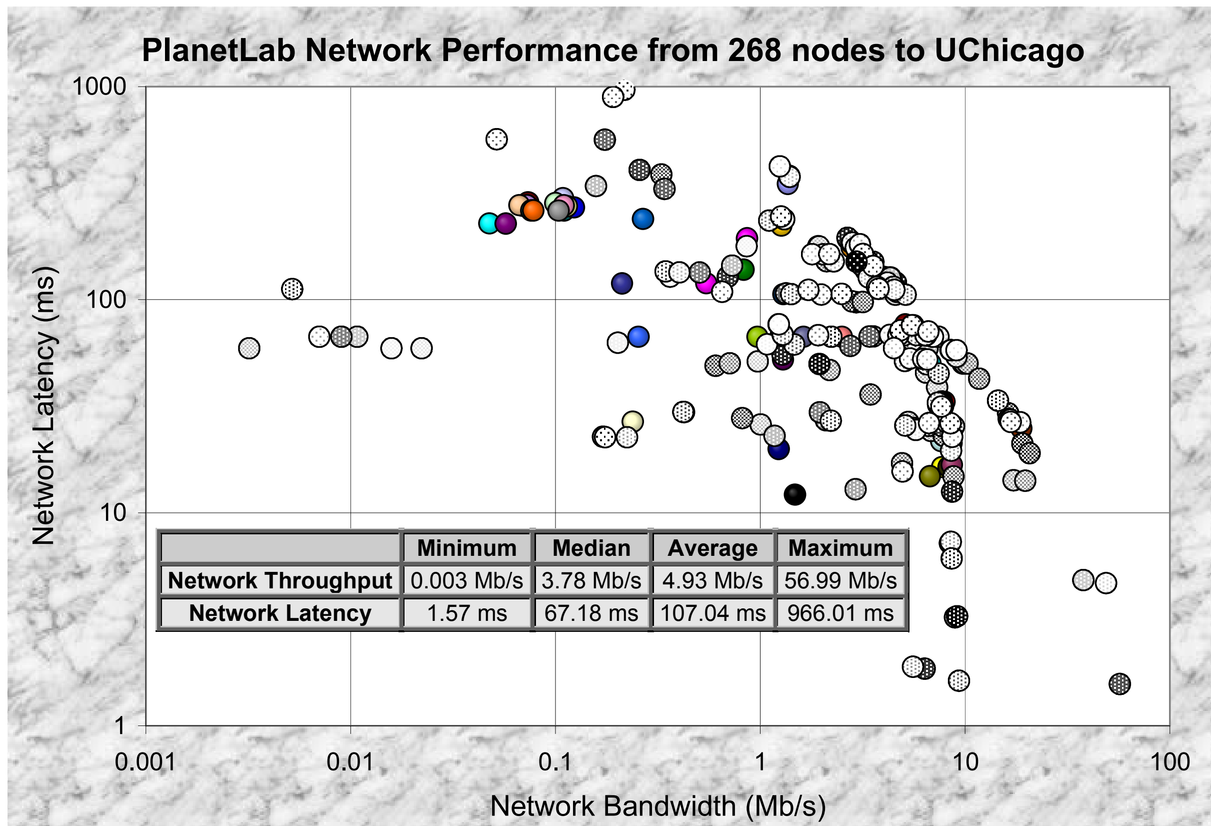
PlanetLab [64] is a geographically distributed platform for deploying, evaluating, and accessing planetary-scale network services. PlanetLab is a shared community effort by a large international group of researchers, each of whom gets access to one or more isolated "slices" of PlanetLab's global resources via a concept called distributed virtualization. In order to encourage innovation in infrastructure, PlanetLab decouples the operating system running on each node from a set of multiple, possibly 3rd-party network-wide services that define PlanetLab, a principle referred to as unbundled management.



	Minimum	Median	Average	Maximum
Network Throughput	0.003 Mb/s	3.78 Mb/s	4.93 Mb/s	56.99 Mb/s
Network Latency	1.57 ms	67.18 ms	107.04 ms	966.01 ms

Figure 2: PlanetLab Network Performance from 268 nodes to a node at UChicago as measured by IPERF on April 13th, 2005; each circle denotes a physical machine with the corresponding x-axis and y-axis values as its network characteristics, namely network latency and bandwidth.

PlanetLab's deployment is now at over 500 nodes (Linux-based PCs or servers connected to the PlanetLab overlay network) distributed around the world. Almost all nodes in PlanetLab are connected via 10 Mb/s network links (with 100Mb/s on several nodes), have processors speeds exceeding 1.0 GHz IA32 PIII class processor, and at least 512 MB RAM. Due to the large geographic distribution (the entire world) among PlanetLab nodes, network latencies and achieved bandwidth varies greatly from node to node. In order to capture this variation in network performance, Figure 2 displays the network performance of 268 nodes (the accessible nodes on 04-13-05) as measured by IPERF on April 13th, 2005. It is very interesting to note the heavy dependency between high bandwidth / low latencies and low bandwidth / high latencies. In order to visualize the majority of the node characteristics better, Figure 3 shows the same data from Figure 2, but with the x and y axis shown at log scale.



	Minimum	Median	Average	Maximum
Network Throughput	0.003 Mb/s	3.78 Mb/s	4.93 Mb/s	56.99 Mb/s
Network Latency	1.57 ms	67.18 ms	107.04 ms	966.01 ms

Figure 3: PlanetLab Network Performance from 268 nodes to a node at UChicago as measured by IPERF on April 13th, 2005 shown with x and y axis in log scale.

2.3.2.3 UChicago CS Cluster

The University of Chicago CS cluster contains over 100 machines that are remotely accessible. The majority of these machines are running Debian Linux 3.0, have AMD Athlon XP Processors at 2.1GHz, have 512 MB of RAM, and are connected via a 100 Mb/s Fast Ethernet switched network. The communication latency between any pair of machines in the CS cluster is on average less than 1 ms, with a few having latencies as high as several ms. Furthermore, all machines share a common file system via NFS (Network File System).

2.3.2.4 DiPerF Cluster

Some tests were performed on a smaller scale LAN that had better network connectivity, specifically 1Gb/s connections via a switch. The network latencies incurred were generally less than 0.1 ms. This cluster did not run NFS as was the case in the UChicago CS cluster. The connectivity of the DiPerF cluster to the outside world is 100Mb/s.

Table 1: DiPerF cluster at UChicago hosts hardware and OS details

Machine Name	m5	diablo	cobra	512tr	viper
Machine Type	x86 64 bit	x86 64 bit	X86 32 bit	X86 32 bit	X86 32 bit
OS	Linux Mandrake 10.1	Linux Suse 9.2	Linux Suse 9.2	Linux Suse 9.2	Linux Mandrake 10.1
OS Release	2.6.8.1-12mdksmp	2.6.8-24-default	2.6.4-52-default	2.6.8-24-default	2.6.8.1-12mdk
# of Proc.	2	1	1	1	1
CPU Speed	1600 MHz	1800 MHz	2166 MHz	2166 MHz	1466 MHz
Cache Size	L1: 256KB L2: 2048KB	L1: 128KB L2: 256KB	L1: 128KB L2: 256KB	L1: 128KB L2: 256KB	L1: 128KB L2: 256KB
CPU Type	AMD Opteron	AMD Athlon 64	AMD Athlon XP	AMD Athlon XP	AMD Athlon XP
Memory Total	1 GB DDR PC3200 Dual Channel	1 GB DDR PC3200 Dual Channel	1 GB DDR PC2700	1 GB DDR PC2700	768 MB SDRAM PC133
Swap Total	1 GB	1 GB	1 GB	1 GB	1 GB
Network Link	1 Gb/s Int. 100 Mb/s Ext.	1 Gb/s Int. 100 Mb/s Ext.	1 Gb/s Int. 100 Mb/s Ext.	1 Gb/s Int. 100 Mb/s Ext.	100 Mb/s Int. 100 Mb/s Ext.
Network MTU	1500 B	1500 B	1500 B	1500 B	1500 B

2.3.2.5 Other Machines

In testing the GridFTP server, we also used some machines at ISI in the NED Cluster. The machine specification of the machine that was used in the GridFTP tests can be found in Table 2.

Table 2: Host specifications of machine used in GridFTP tests located at ISI

Machine Name	ned-6.isi.edu
Machine Type	x86
OS	Linux
OS Release	2.6.8.1-web100
Number of Processors	2
CPU Speed	1126 MHz
Memory Total	1.5 GB
Swap Total	2 GB
Network Link	1 Gb/s Ethernet
Network MTU	1500 B
GridFTP Server Version	0.13 (gcc32dbg, 1103191677-1) Development Release

2.3.3 Communication Protocols: a Layered Approach

Layering is one of the major reasons network architectures have been so successful. One great success story is the Internet, which shows how robust and scalable it has been despite the initial design goals which did not foresee the exponential growth that it endured.

Layering helps break complex problems into smaller more manageable pieces. It helps reduce design complexity and it simplifies the design and testing protocols. Sender and receiver software can be tested, designed and implemented independently. Layering prevents changes in software from propagation to other layers. It allows designers to construct protocol suites and allows ease of change regarding an implementation of a service. Some of its drawbacks include some performance loss, time delay, and perhaps having more than 1 copy of data at any given moment. Obviously, these drawbacks are quickly overshadowed by all the advantages of a layered approach to designing protocols.

The basic definition of layering is that the layer N software on the receiving machine should receive the exact message sent by the layer N software at the sender machine. It should satisfy whatever transformation was applied to the packet should be completely reversible at the receiving side.

The OSI model is not a network architecture because it does not specify exact services and protocols. It is designed for open system interconnection. Each layer should represent a well defined function and a new layer is needed when a new level of abstraction is required. The layers should be chosen in order to minimizing flow of information across layers. And last of all, each layer should be chosen towards standardizing protocols.

Figure 4 depicts the OSI reference model and its 7 layers. To put IP [65], TCP [66], UDP [67] and SSH [68] in perspective, it is relevant to discuss the TCP/IP reference model from Figure 5.

The OSI model is composed of 7 layers:

Layer 1: Application layer
Layer 2: Presentation layer
Layer 3: Session layer
Layer 4: Transport layer
Layer 5: Network layer
Laver 6: Data link laver

Figure 4: OSI Reference Model

The TCP/IP model normally found in practice is composed of 4 layers:

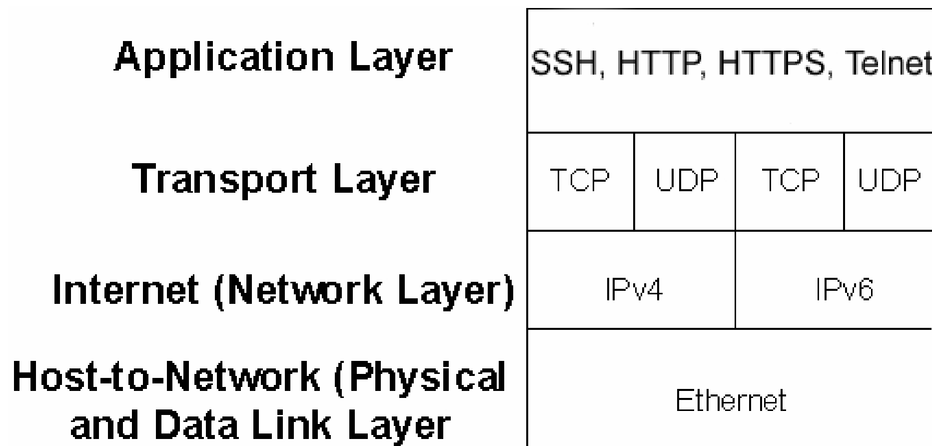


Figure 5: TCP/IP Reference Model; on the left the various levels are identified while on the right examples of functionality/protocol at each respective layer

2.3.3.1 TCP & UDP

The Internet layer, known as the network layer in the OSI model, allows heterogeneous networks to be connected. It provides congestion control, it establishes, maintains, and tears down connections, and most important of all, it determines the route of packets transmitted. Both the IP protocol versions, IPv4 [65] and IPv6 [69, 70], are found in the network layer.

The transport layer provides reliable, transparent data transfers between senders and receivers. It provides error recovery mechanism and flow control in order to throttle the sending rates. It also fragments data into smaller pieces, and passes them down to the network layer. Both TCP [66] and UDP [67] are found in the transport layer.

TCP is a connection oriented protocol, which enables two hosts to establish a connection and exchange streams of data. TCP guarantees delivery of data and also guarantees that packets will be delivered in the same order in which they were sent. Furthermore, TCP performs both congestion control and congestion avoidance through its AIMD (additive increase and multiplicative decrease) algorithm.

On the other hand, UDP is a connectionless protocol that, like TCP, runs on top of IP networks. Unlike TCP/IP, UDP/IP provides very few error recovery services, offering instead a direct way to send and receive datagrams over an IP network. UDP is considered to be a light weight network protocol in contrast to the heavier weight TCP protocol.

The application layer has many protocols used in conjunction with the application. SSH, HTTP, HTTPS, TELNET, FTP, and DNS are only a few that are among the protocols that applications can use which relies on either IP, UDP, or TCP for end-to-end communication.

In summary, UDP is considered to be a lighter weight protocol than TCP, and should in theory be more scalable than TCP due to the fact that UDP is a connectionless protocol. On the other hand, UDP is not reliable, and hence not all application can benefit from UDP's scalability due to strict reliability constraints.

2.3.3.2 SSH

SSH [68] is a protocol for secure remote login and other secure network services over an insecure network. It consists of three major components:

- The Transport Layer Protocol provides server authentication, confidentiality, and integrity. It may optionally also provide compression. The transport layer will typically be run over a TCP/IP connection, but might also be used on top of any other reliable data stream.
- The User Authentication Protocol authenticates the client-side user to the server. It runs over the transport layer protocol.
- The Connection Protocol multiplexes the encrypted tunnel into several logical channels. It runs over the user authentication protocol.

The client sends a service request once a secure transport layer connection has been established. A second service request is sent after user authentication is complete. This allows new protocols to be defined and coexist with the protocols listed above.

The connection protocol provides channels that can be used for a wide range of purposes. Standard methods are provided for setting up secure interactive shell sessions and for forwarding ("tunneling") arbitrary TCP/IP ports and X11 connections.

In summary, SSH is the most expensive communication protocol, due to the overhead of encrypting/decrypting all data that needs to be communicated. SSH is built on top of TCP, and hence it will have at the very least the same scalability issues that TCP will have due to the state that must be maintained. In practice, SSH is less scalable than TCP due to extra processing that must take place.

2.3.4 Test Cases

This section will cover the basics of the various components of the Globus Toolkit, including GridFTP, MDS, and GRAM. It will also cover the basic architecture of GRUBER, a grid service built on top of the Globus Toolkit 3.2.

2.3.4.1 Globus Toolkit 3.2

Globus Toolkit 3 (GT3) is based on a new core infrastructure component compliant with the Open Grid Services Architecture (OGSA) [4], and is an open source implementation of the Open Grid Services Infrastructure (OGSI) [71]. GT3 Core offers a run-time environment capable of hosting Grid services. The run-time environment mediates between the application and the underlying network, and transport protocol engines. GT3 Core also provides development support including programming models for exposing, and accessing Grid service implementations.

The main design goal of GT3 has been to make the OGSI technology easy to use, reuse, and extend when developing new Grid applications. The GT3 Core also provides various hosting environments built around a container abstraction. The container enables portable OGSI compliant Grid services to be developed without any knowledge of the underlying protocols and transport bindings. The GT3 Core can be seen as the set of building blocks we consider essential for all Grid applications. The OGSI primitives implemented offer support for soft-state management, inspection, notification, discovery and global instance naming. Additionally, GT3 Core is comprised of a security infrastructure, and a number of system-level services, such as logging-, management-, and administration Grid services. Figure 6 depicts the system overview of the Globus Toolkit 3.2. [72]

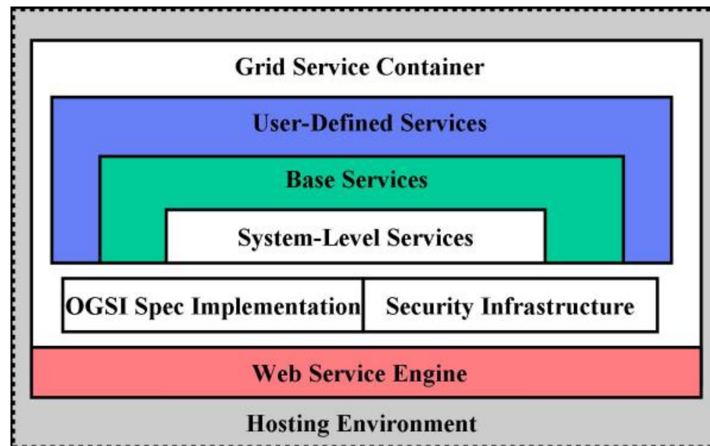


Figure 6: Globus Toolkit System Overview [72]

2.3.4.2 Globus Toolkit 3.9.5

The Globus Toolkit 3.9.5 is in essence the Beta version of GT4 released in April 2005; since all the experiments were conducted prior to this date, all results are based on pre-GT4 releases. As shown in Figure 7, GT4 comprises both a set of service implementations ("server" code) and associated "client" libraries. GT4 provides both Web services (WS) components (on the top) and non-WS components (on the bottom). Note that all GT4 WS components use WS-Interoperability-compliant transport and security mechanisms, and can thus interoperate with each other and with other WS components. In addition, all GT4 components, both WS and non-WS, support X.509 end entity certificates and proxy certificates. Thus a client can use the same credentials to authenticate with any GT4 WS or non-WS component.

Nine GT4 services implement Web services (WS) interfaces:

- job management (GRAM)
- reliable file transfer (RFT)

- delegation
- Monitoring and Discovery System (MDS)
    - MDS-Index
    - MDS-Trigger
    - MDS Aggregator
- community authorization (CAS)
- OGSA-DAI data access and integration
- GTCP Grid TeleControl Protocol for online control of instrumentation.

For two of these services, GRAM and MDS-Index, pre-WS "legacy" implementations are also provided. These pre-WS implementations will be deprecated at some future time as experience is gained with WS implementations. For three additional GT4 services, WS interfaces are not yet provided (but will be in the future):

- GridFTP data transport
- replica location service (RLS), and
- MyProxy online credential repository

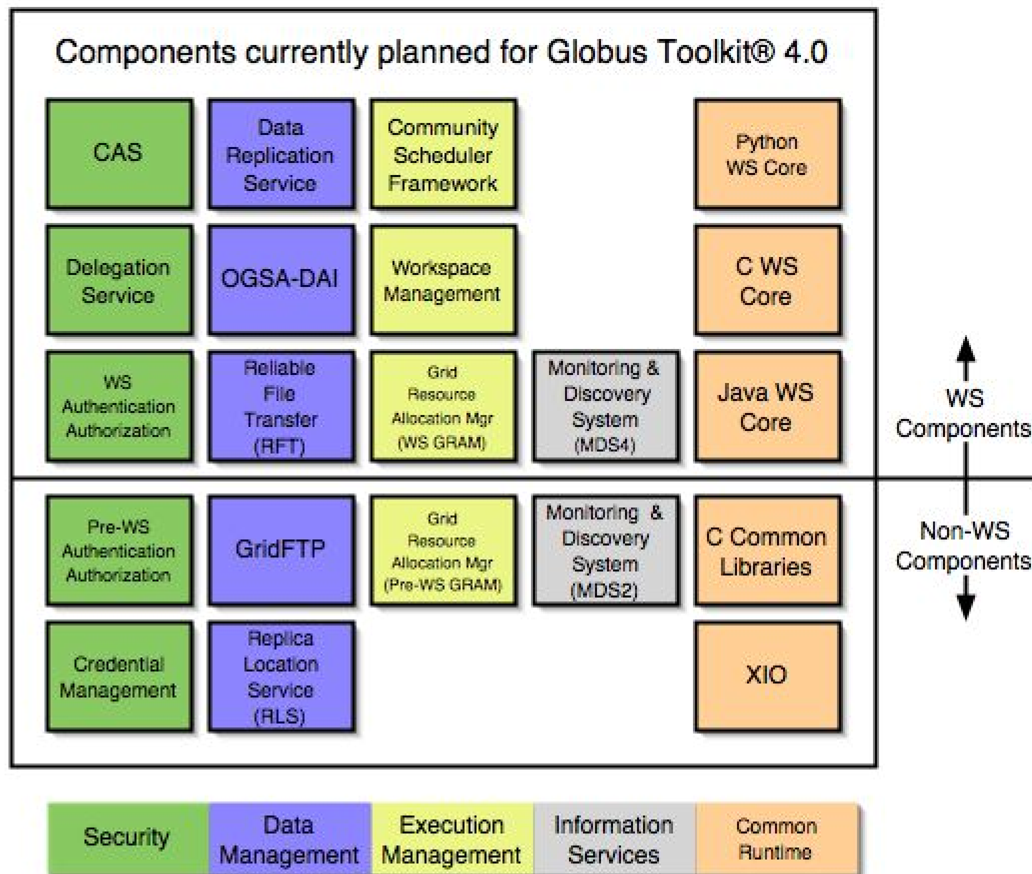


Figure 7: GT4 Key Components [73]

Other libraries provide powerful authentication and authorization mechanisms, while the eXtensible I/O (XIO) library provides convenient access to a variety of underlying transport protocols. SimpleCA is a lightweight certification authority.

2.3.4.3 GRAM

GRAM (Grid Resource Allocation and Management) simplifies the use of remote systems by providing a single standard interface for requesting and using remote system resources for the execution of "jobs". The most common use (and the best supported use) of GRAM is remote job submission and control. This is typically used to support distributed computing applications.

GRAM is designed to provide a single common protocol and API for requesting and using remote system resources, by providing a uniform, flexible interface to local job scheduling systems. The Grid Security Infrastructure (GSI) provides mutual authentication of both users and remote resources using GSI (Grid-wide) PKI-based identities. GRAM provides a simple authorization mechanism based on GSI identities and a mechanism to map GSI identities to local user accounts. [74]

We evaluated three implementations of a job submission service bundled with various versions of the Globus Toolkit:

- GT3.2 pre-WS GRAM
- GT3.2 WS GRAM
- GT4 WS GRAM.

GT3.2 pre-WS GRAM performs the following steps for job submission: a gatekeeper listens for job requests on a specific machine; performs mutual authentication by confirming the user's identity, and proving its identity to the user; starts a job manager process as the local user corresponding to authenticated remote user; then the job manager invokes the appropriate local site resource manager for job execution and maintains a HTTPS channel for information exchange with the remote user.

GT3.2 WS GRAM, a WS-based job submission service, performs the following steps: a client submits a *createService* request which is received by the Virtual Host Environment Redirector, which then attempts to forward the *createService* call to a User Hosting Environment (UHE) where mutual authentication / authorization can take place; if the UHE is not created, the Launch UHE module is invoked; WS GRAM then creates a new Managed Job Service (MJS); MJS submits the job into a back-end scheduling system [75].

GT3.9.5 WS-GRAM models jobs as lightweight WS-Resources rather than relatively heavyweight Grid services. WS GRAM combines job-management services and local system adapters with other service components of GT 4.0 in order to support job execution with coordinated file staging. Figure 8 depicts the complex set of message exchanges that occurs in WS-GRAM. We note that both pre-WS GRAM and WS GRAM are complex services: a job submission, execution, and result retrieval sequence may include multiple message exchanges between the submitting client and the service. [76]

The heart of the WS GRAM service architecture is a set of Web services designed to be hosted in the Globus Toolkit's WSRF core hosting environment. Each submitted job is exposed as a distinct resource qualifying the generic ManagedJob service. The service provides an interface to monitor the status of the job or to terminate the job (by terminating the ManagedJob resource). Each compute element, as accessed through a local scheduler, is exposed as a distinct resource qualifying the generic ManagedJobFactory service. The service provides an interface to create ManagedJob resources of the appropriate type in order to perform a job in that local scheduler.

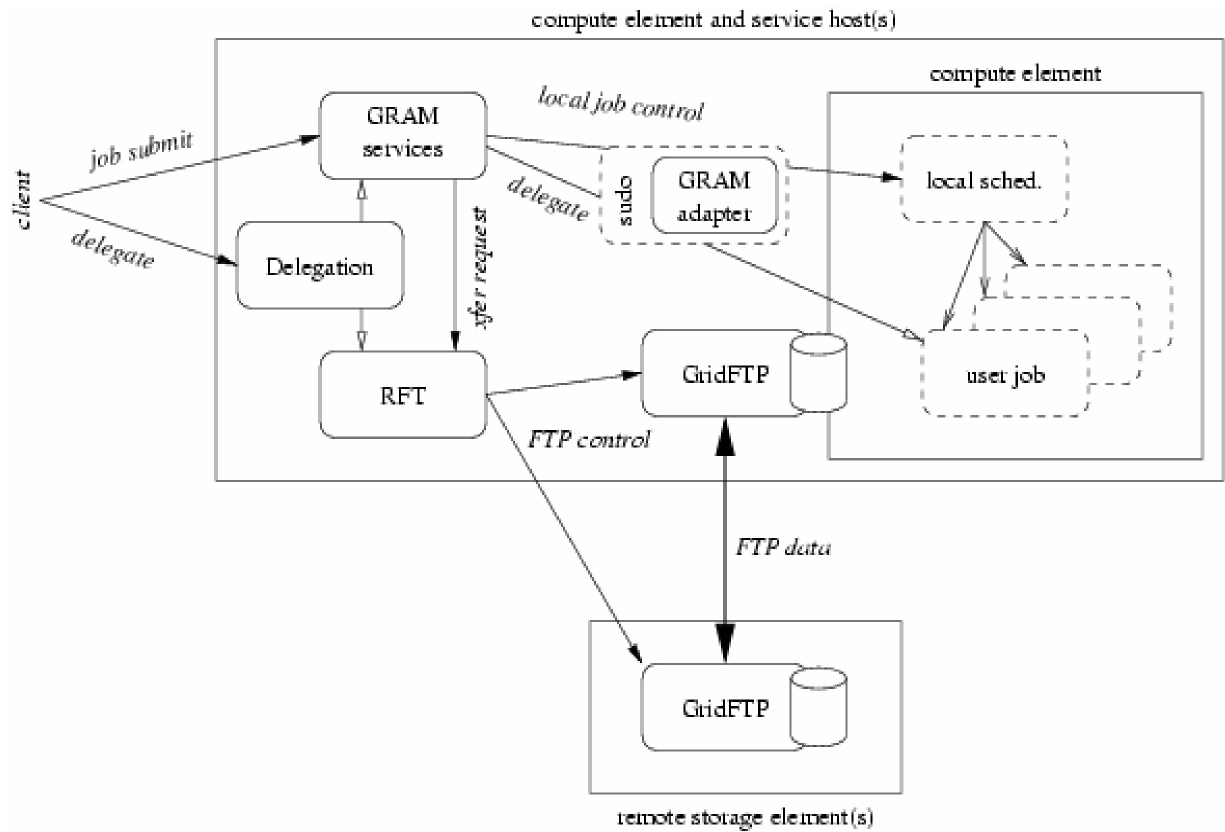


Figure 8: WS GRAM Component Architecture Approach [76]

At a high level, we can consider the main client activities around a WS GRAM job to be a partially ordered sequence as depicted in Figure 9. [76]

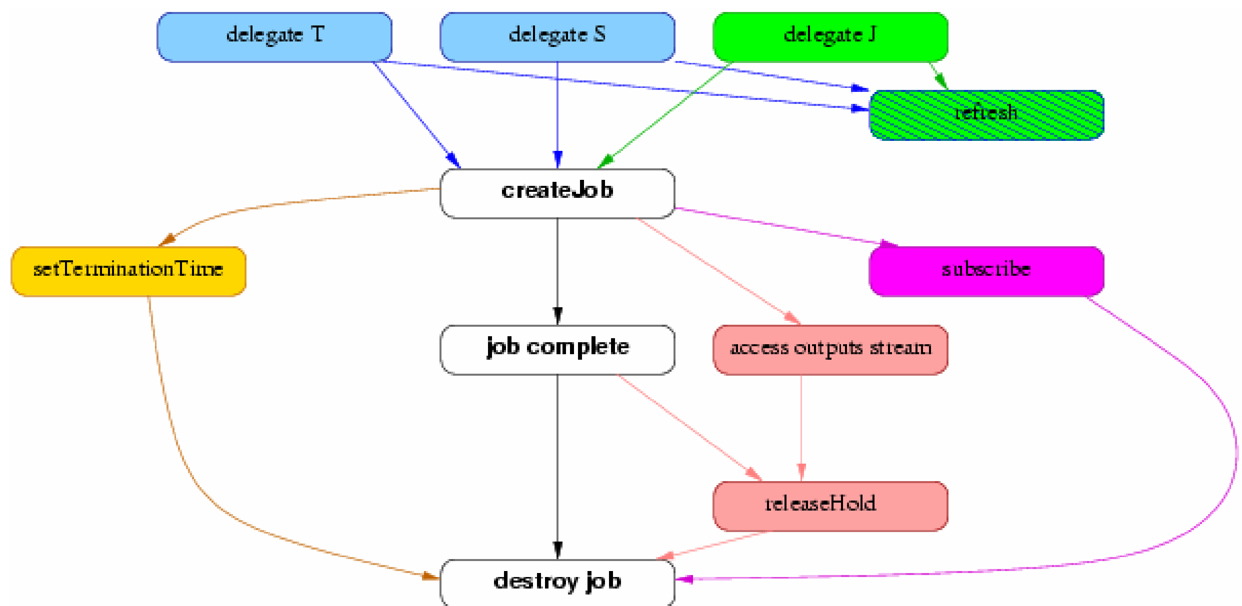


Figure 9: Partially ordered sequence of client activities around a WS GRAM job [76]

2.3.4.4 GridFTP

GridFTP is a high-performance, secure, reliable data transfer protocol optimized for high-bandwidth wide-area networks. The GridFTP protocol is based on FTP, the highly-popular Internet file transfer protocol. We have selected a set of protocol features and extensions defined already in IETF RFCs and added a few additional features to meet requirements from current data grid projects. The white paper, "GridFTP: Universal Data Transfer for the Grid" [77], describes the motivations behind this work in more detail. Some of the protocol features include:

- GSI security on control and data channels
- Multiple data channels for parallel transfers
- Partial file transfers
- Third-party (direct server-to-server) transfers
- Authenticated data channels
- Reusable data channels
- Command pipelining

The implementation of the GridFTP protocol takes the form of two APIs and corresponding libraries: globus_ftp_control and globus_ftp_client. The libraries use high-performance I/O and security services provided by the Globus Toolkit. There also exists an API/library (globus_gass_copy) and a command-line tool (globus-url-copy) that integrates GridFTP, HTTP, and local file I/O to enable secure transfers using any combination of these protocols. Finally, a popular FTP server package (Washington University's wu-ftpd) has been adapted to support a majority of the GridFTP protocol features (GSI security, parallel transfer, third-party transfer, partial file transfer).

2.3.4.5 WS-MDS

The Monitoring and Discovery System (MDS) is composed of three components: MDS-Index, MDS-Trigger, and the MDS Aggregator. [78]

The Index Service collects monitoring and discovery information from Grid resources, and publishes it in a single location; generally, it is expected that a virtual organization will deploy one or more index services which will collect data on all of the Grid resources available within that virtual organization.

The Trigger Service collects data from resources on the grid and, if administrator defined rules match, can perform various actions. An example use is to send email when queue length on a compute resource goes over a threshold value.

The WS MDS Aggregator is the software framework on which WS MDS services (currently, the WS MDS Index and WS MDS Trigger services) are built. The aggregator framework collects data from an aggregator source and sends that data to an aggregator sink for processing. Aggregator sources distributed with the Globus Toolkit include modules that query service data, acquire data through subscription/notification, and execute programs to generate data. Aggregator sinks include modules that implement the WS MDS Index service interface and the WS MDS Trigger service interface.

2.3.4.6 GRUBER & DI-GRUBER

Resource sharing within grid collaborations usually implies specific sharing mechanisms at participating sites. Challenging policy issues can arise in such scenarios that integrate participants and resources spanning multiple physical institutions. Resource owners may wish to grant to one or more virtual organizations (VOs) the right to use certain resources subject to local usage policies and service level agreements, and each VO may then wish to use those resources subject to its usage policies. GRUBER is an architecture and toolkit for resource usage service level agreement (SLA) specification and enforcement in a grid environment. The proposed mechanism allows resources at individual sites to be shared among multiple user communities. GRUBER ultimately

addresses issues regarding how usage SLAs can be stored, retrieved and disseminated efficiently in a large distributed environment.

The environment model which was used for the evaluation and experimentation is depicted in Figure 10 [79, 80]. The main elements of this work are the *decision points* (previously known as policy enforcement points or PEPs), which are responsible for executing SLAs. They gather monitoring metrics and other information relevant to their operations, and then use this information to steer resource allocations as specified by the usage SLAs [81].

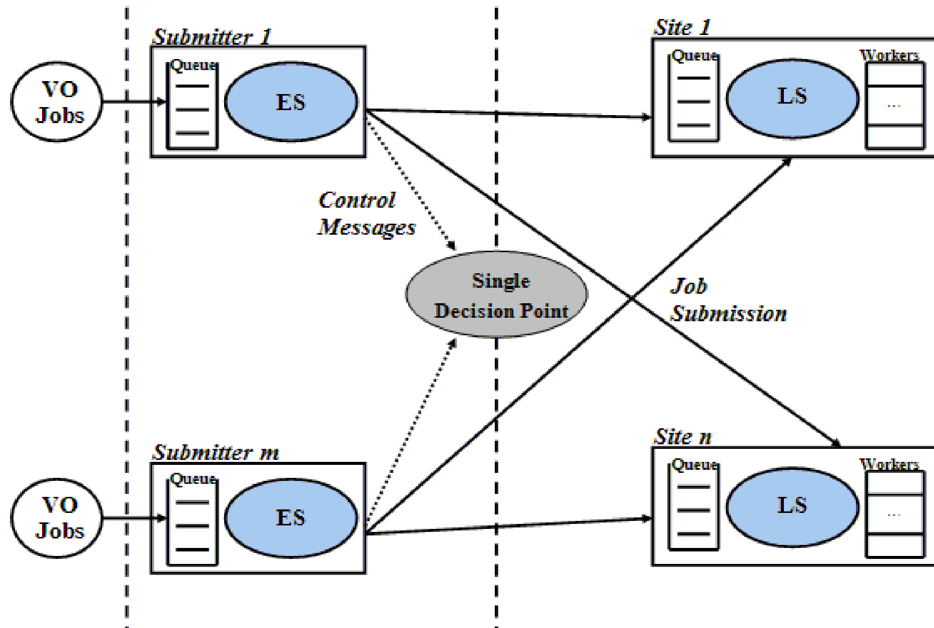


Figure 10: VO-Level Architecture

GRUBER makes the assumption that each decision point has full "static" knowledge about available resources for its users, but not the latest resource utilizations. Practically, each decision point relies on information exchanges for updating only its view on the current utilizations. Another advantage of this approach is that it simplifies the implementation greatly, by avoiding the tracking of each usage SLA and allocation apparition time, as well as the entity to which it applies.

3 DiPerF Framework

DiPerF [82] is a DIstributed PERformance testing Framework aimed at simplifying and automating service performance evaluation. DiPerF coordinates a pool of machines that test a single or distributed target service, collects and aggregates performance metrics from the client point of view, and generates performance statistics. The aggregate data collected provides information on service throughput, service response time, on service 'fairness' when serving multiple clients concurrently, and on the impact of network latency on service performance. All steps involved in this process are automated, including dynamic deployment of a service and its clients, testing, data collection, and data analysis.

Figure 11 shows an example screenshot of a demo showing off the DiPerF from beginning to end. The screenshot below shows the end of the experiment in which the data analysis is complete and the graph with both the client and server view of the results have been generated; the GKrellM [83] Monitor is used to show the hosts (server and controller) performance in terms of CPU utilization, network transfer rates, and hard disk throughput.

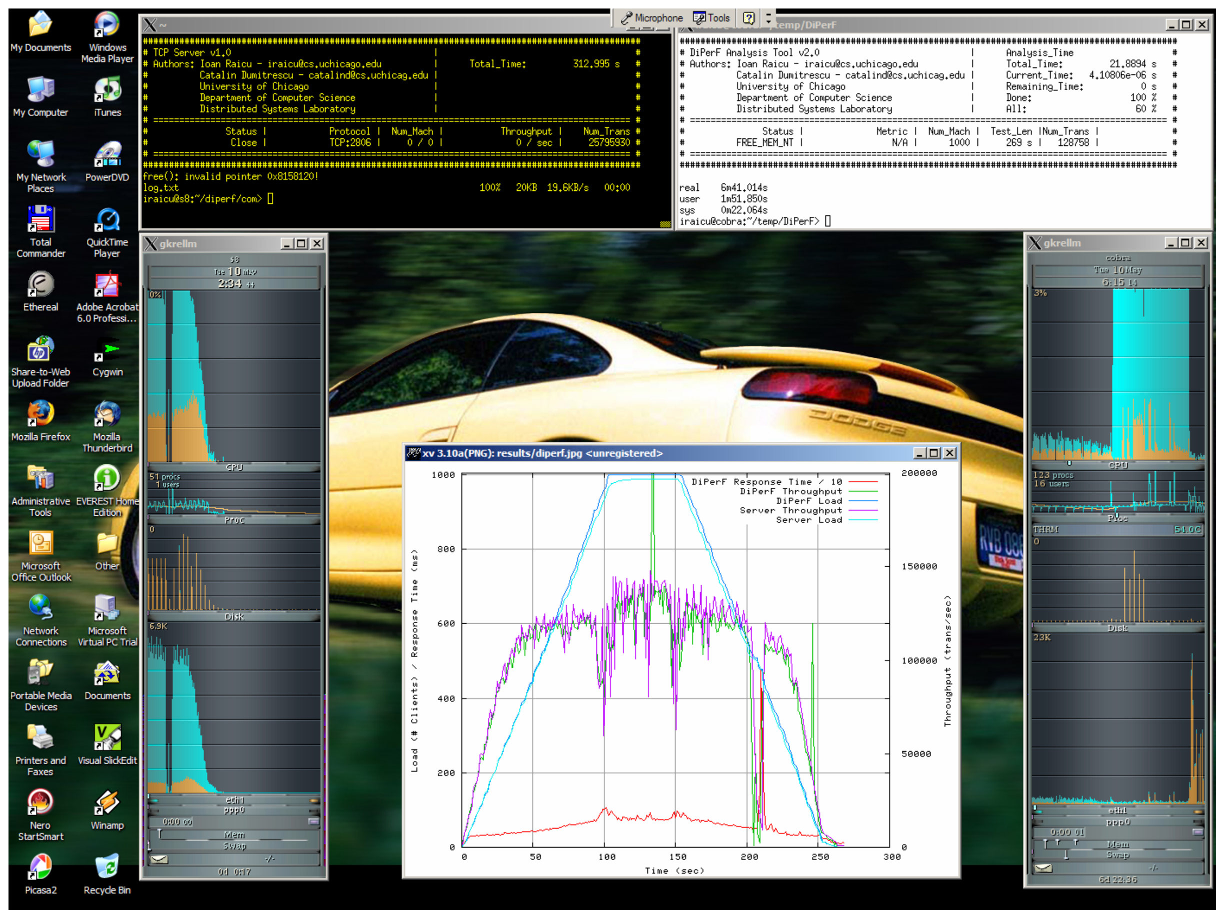


Figure 11: DiPerF demo showing the end of an experiment with 1000 clients accessing a TCP server

DiPerF consists of two major components: the *controller* and the *testers* (Figure 12). A user of the framework provides to the controller the address or addresses of the target service to be evaluated and the client code for the service. The controller starts and coordinates a performance evaluation experiment: it receives the client code, distributes it to testers, coordinates their activity, collects and finally aggregates their performance measurements. Each tester runs the client code on its machine, and times the (RPC-like) network calls this code

makes to the target service. Finally, the controller collects all the measurements from the testers and performs additional operations (e.g., reconciling time stamps from various testers) to compute aggregated performance views.

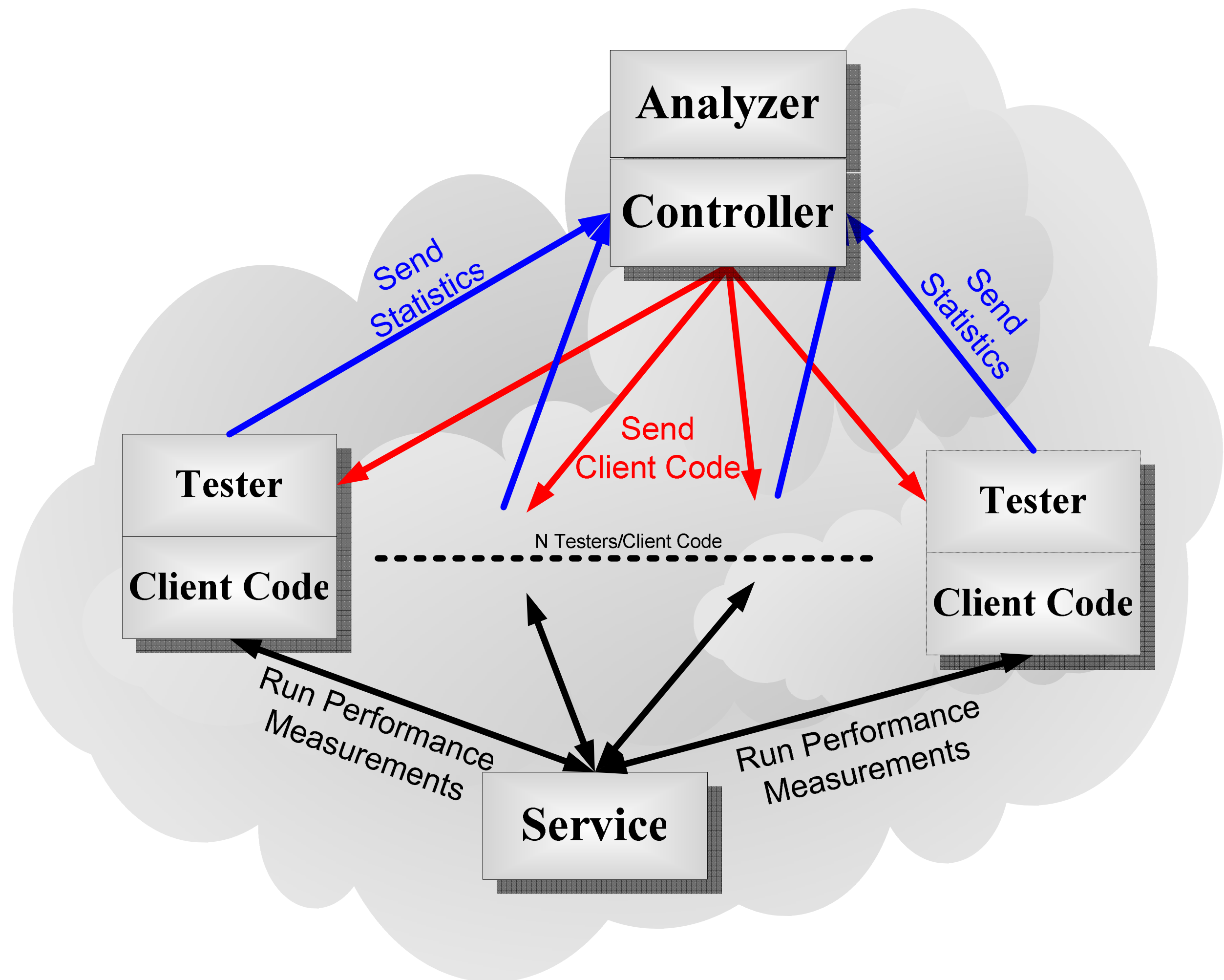


Figure 12: DiPerF framework overview

Figure 13 depicts an overview of the deployment of DiPerF in different testbeds (PlanetLab, UChicago CS cluster, and Grid3). Note the different client deployment mechanisms between the different testbeds, with GT GRAM based submission for Grid3 and ssh based for the other testbeds. Another interesting difference between Grid3 and the other testbeds is the fact that the controller only communicates with a resource manager, and it is the resource manager's job to deploy and launch the tester/client code on physical machines in Grid3; in the other testbeds, the controller is directly responsible of having a complete list of all machines in the testbed and the communication is directly between the remote machine and the controller.

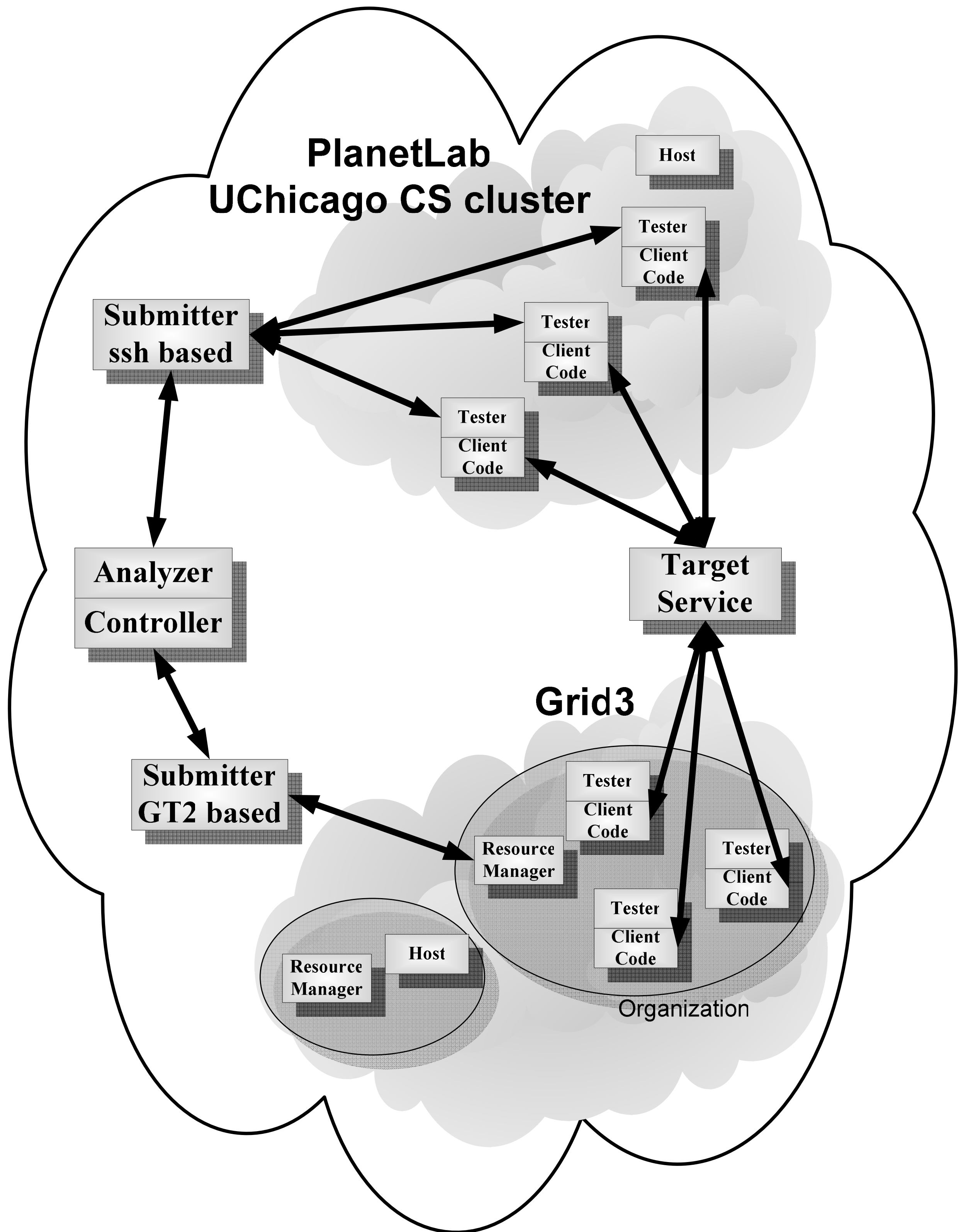


Figure 13: DiPerF framework overview deployment scenario

The interface between the tester and the client code can be defined in a number of ways (e.g., by using library calls); we take what we believe is the most generic avenue: clients are full blown executables that make one RPC-like call to the service. If multiple calls are to be made in each execution of the executable (i.e. when just starting the executable is expensive as is the case in many Java programs), a more sophisticated interface can be designed where the tester gets periodic information from the client code in a predefined format.

The framework is supplied with a set of candidate nodes for client placement, and selects those available as testers. In future work, we will extend the framework to select a subset of available tester nodes to satisfy specific requirements in terms of link bandwidth, latency, compute power, available memory, and/or processor load. In its current version, DiPerF assumes that the target service is already deployed and running.

Some metrics are collected directly by the testers (e.g., response time), while others are computed at the controller (e.g., throughput and service fairness). Additional metrics (e.g. network related metrics such as network throughput, size of data transmitted, time to complete a subtask, etc), measured by clients can be reported, through an additional interface, to the testers and eventually back to controller for statistical analysis. Testers send performance data to the controller while the test is progressing, and hence the service evolution and performance can be visualized 'on-line'.

Communication among DiPerF components has been implemented with several flavors: ssh based, TCP, and UDP. When a client fails, we rely on the underlying protocols (i.e. whatever the client uses such as TCP, UDP, HTTP, pre-WS GRAM, etc) to signal an error which is captured by the tester which is in turn sent to the controller to delete the client from the list of the performance metric reporters. A client could fail because of various reasons: 1) predefined timeout which the tester enforces, 2) client fails to start (i.e. out of memory - OS client machine related), 3) and service denied or service not found (service machine related). Once the tester is disconnected from the controller, it stops the testing process to avoid loading the target service with requests which will not be aggregated in the final results.

3.1 Scalability Issues

Our initial implementation [82] of DiPerF as it appeared at Grid2004 was scalable, but it could be improved, and hence we made several improvements to increase the scalability and performance of DiPerF. Based on the implementation using the ssh based communication protocol, DiPerF was limited to only about 500 clients. We therefore concentrated on reducing the amount of processing per transaction, reducing the memory footprint of the controller, and reducing the number of processes being spawned in relation to the number of desired testers/clients throughout the experiment.

In order to make DiPerF as flexible as possible for a wide range of configurations, we stripped down the controller from most of its data processing tasks (which are online) and moved them to the data analysis component (which is offline); this change helped the controller be more scalable, freeing the CPU of unnecessary load throughout the experiment. If it is desirable for the results to be viewed in real-time as the experiment progresses, then there exists another version of the controller that is more complex, but will give the user feedback of the performance in real-time. Furthermore, for increased flexibility, the controller can work in two modes: write data directly to the hard disk, or keep data in memory for faster analysis later and reduced load due to the fact that it does not have to write to the disk except when the experiment is over.

We also added the support of multiple testers on the same node by identifying a tester by the node name followed by its process ID (pid); this feature helped in managing multiple testers on the same node, which allows the testing of services beyond the size of the testbed. For example, if using a 100 physical node testbed, and having 10 testers/clients on each node, it would add up to 1000 testers/clients. This method of increasing the number of clients by running multiple clients on each physical node does not work for any client. If the client is heavy weight, and requires significant amounts of resources from the node, the average client performance will decrease as the number of clients increases; note that heavy weight clients will most likely produce inaccurate server performance once the number of clients surpasses the number of physical machines. On the other hand, this is a nice feature to have because it makes scalability studies for any client/service possible even with a small number of physical machines.

To alleviate the biggest bottleneck that we could identify, namely the communication based on ssh, we implemented two other communication protocols on top of TCP and UDP. Running TCP will allow us to have the same benefits of ssh (reliability), but will have less overhead because the information is not encrypted and decrypted, a relatively expensive operation. Using sockets, we can also control the buffer sizes of the connections more easily (without root privileges), and hence get better utilization of the memory of the system, especially since we can sacrifice buffer size without affecting performance due to the low needed bandwidth per connection. Using UDP, we get a stateless implementation, which will be much more scalable than any TCP or ssh based implementation. We get all the advantages of TCP (as mentioned above), but we loose the reliability; simple and relatively inexpensive methods to ensure some level of reliability could be implemented on top of UDP.

Finally, in order to achieve the best performance with the implementation of the communication over TCP or UDP, we used a single process which used the select() system function [84] to provide synchronous I/O multiplexing between 1,000s of concurrent connections. The functions select() wait for a number of file descriptors (found in fd_set) to change status based on a timeout value specified in the number of seconds and microseconds. First of all, the fd_set is a fixed size buffer as defined in several system header files; most Linux based systems have a fixed size of 1024. This means that any given select() function can only have 1024 file descriptors (i.e. TCP sockets) that it is listening on. This is a huge limitation, and would limit any 1 process implementation over TCP to only 1024 clients. After modifying some system files (required root access) to raise the constant size fd_set from 1024 to 65536, we were able to break the 1024 concurrent client barrier. However, we now had another issue to resolve, namely the expensive operation of initializing the fd_set (one file descriptor at a time) every time a complete pass through the entire fd_set; with an fd_set size of 1024, this did not seem to be a problem, but with an fd_set size of 65536, it quickly became a bottleneck. The solution we employed was to keep two copies of the fd_set, and after a complete pass through all the entire fd_set, simply do a memory to memory copy from one fd_set to another, a significantly less expensive operation than having to reset the fd_set one file descriptor at a time. We are currently working on porting the select()-based implementation to a /dev/poll-based [85] implementation that is considered to be significantly lighter weight than select() with less of an overhead.

With all these performance and scalability improvements, DiPerF can now scale from 4,000 clients using ssh to 60,000 clients using TCP to 80,000 clients using UDP; the achieved throughput varied from 1,000 to 230,000 transactions per second depending on the number concurrent clients and the communication protocol utilized. We expect DiPerF's scalability to increase even more once we replace the select() mechanism with /dev/poll; furthermore, we expect DiPerF's achieved throughput to increase even more under high concurrency with 10,000+ clients.

3.2 Client Code Distribution

The mechanisms used to distribute client code (e.g., scp, gsi-scp, or gass-server) vary with the deployment environment. Since ssh-family utilities are deployed on just about any Linux/Unix, we base our distribution system on scp-like tools. DiPerF specifically uses rsync to deploy client code in a Unix-like environment (i.e. PlanetLab, UChicago CS cluster, DiPerF cluster), and it uses GT2 GRAM job submission to deploy client code in a grid environment (i.e. Grid3).

3.3 Clock Synchronization

DiPerF relies heavily on time synchronization when aggregating results at the controller; therefore, an automatic time synchronization among all clients is integrated into DiPerF to ensure that the final results are accurate. Synchronization need not be performed on-line; instead, we can compute the offset between local and global time and apply that offset when analyzing aggregated metrics, assuming that the clock drift over the course of an experiment is not significant. The solution to clock drift is to perform time synchronization on-line at regular intervals that are short enough for the drift to be negligible.

Several off-the-shelf options are available to synchronize the time between machines; one example is NTP [86], which some PlanetLab nodes use to synchronize their time. In a previous study [86], hosts had a mean delay of 33ms, median 32ms, and a standard deviation 115ms from their peer hosts used to synchronize their time with

NTP. These results seem very promising, but unfortunately, at the time that we performed our experiments, we found that most of the nodes in our testbed on PlanetLab were not very well synchronized, with some nodes having synchronization differences in the thousands of seconds. Therefore DiPerF assumes the worst: no clock synchronization mechanism is provided by the deployment platform. To ensure that a mechanism exists to synchronize time among all nodes within tens of milliseconds accuracy, we implemented a timer component that allows all nodes participating in an experiment to query for a 'global' time.

DiPerF handles time synchronization with a centralized time-stamp server that allows time mapping to a common base. The time-stamp server is lightweight and can easily handle 1000+ concurrent clients. In order to avoid clock drift, each client synchronize its clock every five minutes; due to the lightweight time server and relatively rare time synchronization (every 300 seconds), we estimate that it could handle 1,000,000+.

We have measured the latency from over 100 clients (deployed in the PlanetLab testbed) to our timestamp server at UChicago over a period of almost 2 hours. During this interval that the (per-node) latency in the network remained fairly constant and the majority of the clients had a network latency of less than 80ms. The accuracy of the synchronization mechanism we implemented is directly correlated with the network latency and its variance, and in the worst case (non-symmetrical network routes), the timer can be off by at most the network latency. Using our custom synchronization component, we observed a mean of 62ms, median 57ms, and a standard deviation 52ms for the time skew between nodes in our PlanetLab testbed. See Figure 2 for a visual representation of the network performance of the PlanetLab testbed.

Given that the response time of the relatively heavy weight services (i.e. GRAM, GridFTP, MDS, GRUBER) that have been evaluated in this paper are at least one order of magnitude larger, we believe the clock synchronization technique implemented does not distort the results presented.

3.4 Client Control and Performance Metric Aggregation

The controller starts each tester with a predefined delay (specified in a configuration file when the controller is started) in order to gradually build up the load on the service as can be visualized in Figure 14. A tester understands a simple description of the tests it has to perform. The controller sends test descriptions when it starts a tester. The most important description parameters are: the duration of the test experiment, the time interval between two concurrent client invocations, the time interval between two clock synchronizations, and the local command that has to be invoked to run the client. The controller also specifies the addresses of the time synchronization server and the target service.

Individual testers collect service response times. The controller's job is to aggregate these service response times, correlate them with the offered load and with the start/stop time of each tester and infer service throughput, and service 'fairness' among concurrent clients.

Since all metrics collected share a global time-stamp, it becomes simple to combine all metrics in well defined time quanta (seconds, minutes, etc) to obtain an aggregate view of service performance at any level of detail that is coarser than the collected data, an essential feature for summarizing results containing millions of transactions over short time intervals. This data analysis is completely automated (including graph generation) at the user-specified time granularity.

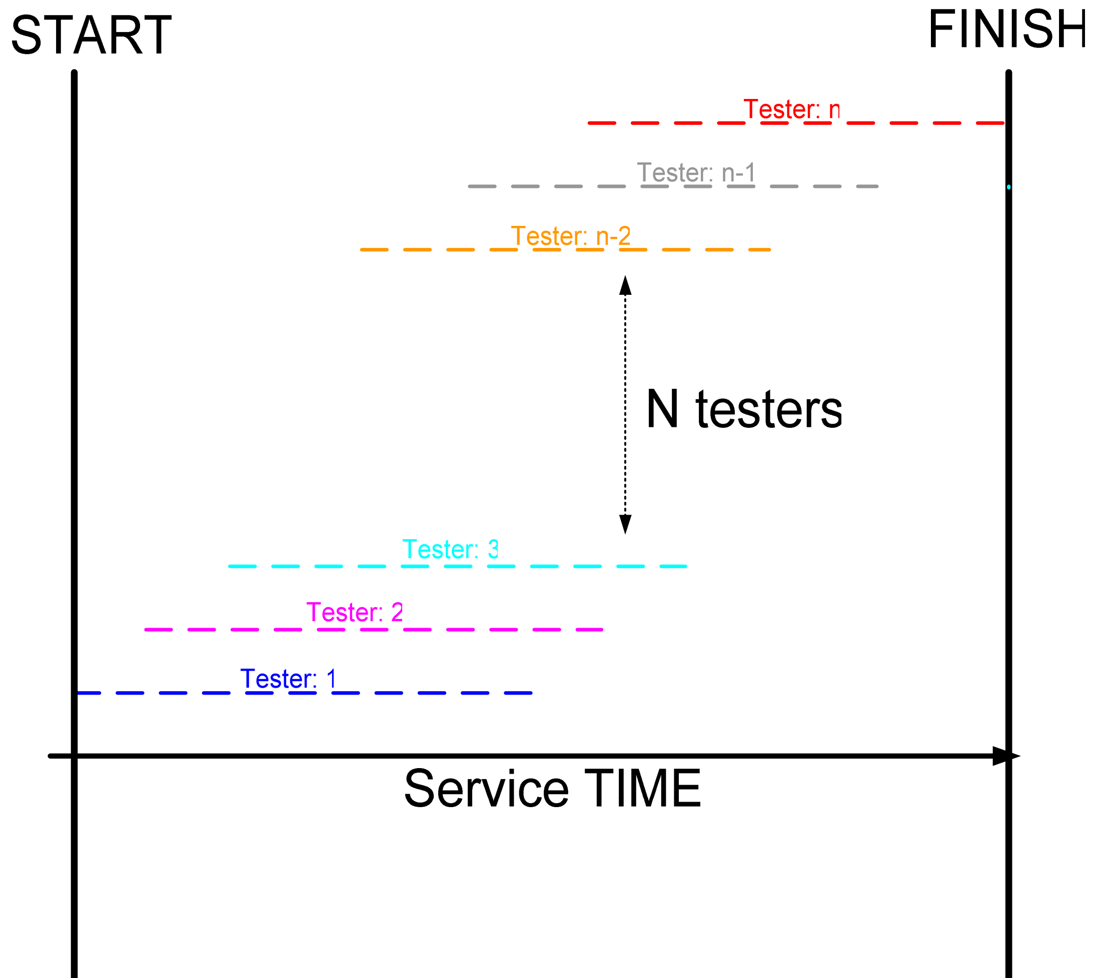


Figure 14: Aggregate view at the controller. Each tester synchronizes its clock with the time server every five minutes. The figure depicts an aggregate view of the controller of all concurrent testers.

3.5 Performance Analyzer

The performance analyzer has been implemented in C++ and currently consists of over 4,000 lines of code. Its current implementation assumes that all performance data is available for processing, which means it is an off-line process; in the future, we plan to port the current performance analyzer to support on-line analysis of incoming performance data. The implementation's main goal has been its flexibility in handling large data analysis tasks completely unsupervised. Furthermore, the performance analyzer was designed to allow a reduction of the raw performance data to a summary of the performance data with samples computed at a specified time quantum. For example, a particular experiment could have accumulated 1,000,000s of performance samples over a period of 3600 seconds, but after the performance analyzer summarizes the data for one sample per second, the end result is reduced to 3600 samples rather than 1,000,000s of samples.

The generic metrics summarized by the performance analyzer based on the user specified time quantum are:

- *service response time* or time to serve a request, that is, the time from when a client issues a request to when the request is completed minus the network latency and minus the execution time of the client code; this metric is measured from the point of view of the client
- *service throughput*: number of jobs completed successfully by the service averaged over a short time interval that is specified by the user (i.e. 1 second, 1 minute, etc) in order to reduce the large number of

samples; to make the results easier to understand, most of the graphs showing throughput also use box plot (moving averages) in order to smooth the throughput metrics out

- *offered load*: number of concurrent service requests (per second)
- *service utilization* (per client): ratio between the number of requests completed for a client and the total number of requests completed by the service during the time the client was active
- *service fairness* (per client): ratio between the number of jobs completed and service utilization
- *job success* (per client): the number of jobs that were successfully completed for a particular client
- *job fail* (per client): the number of jobs that failed for a particular client
- *network latency* (per client): the round trip network latency from the client to the service as measured by the "ping" utility

Among the many performance metrics it can extract from the raw performance data, it has a few additional features. First of all, it has a verify option that allows a user to double check that the raw input data conforms to the correct formatting requirements, and fixes (mostly by deleting) any inconsistencies it finds. Furthermore, all of the above metrics that are computed per client can also be computed over the peak portion of the experiment, when all clients are concurrently accessing the service. This is an important feature for computing the service fairness.

The output resulting from the performance analyzer can be automatically graphed using gnuplot [87] for ease of inspection, and the output can easily be manipulated in order to generate complex graphs combining various metrics such as the ones found in this thesis.

Figure 15 depicts a sample output from the automated gnuplot based on the summarization of the performance analyzer of the achieved throughput of a particular service. The original number of individual performance samples was over 75,000 while the resulting summary had fewer than 700 samples. On the other hand, Figure 16 shows the same graph manually made that contains the throughput (right axis) and two other metrics (load and response time) superimposed.

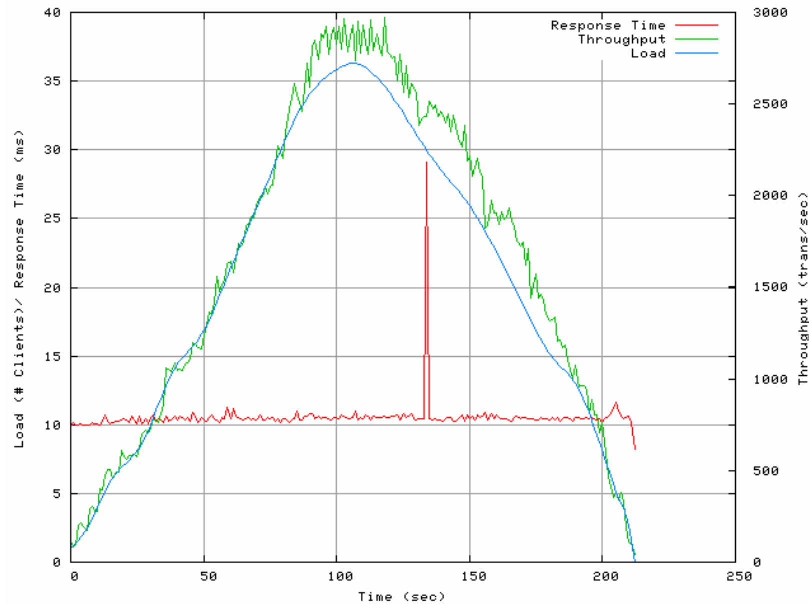
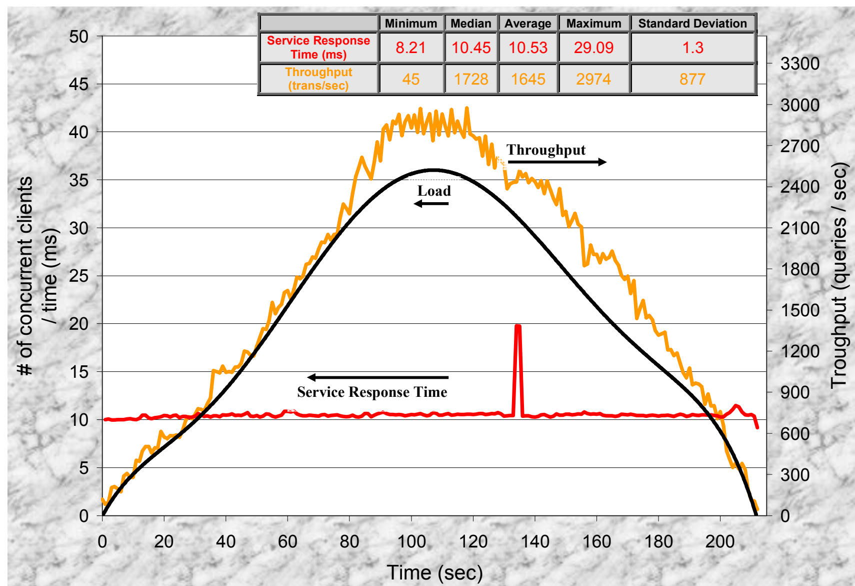


Figure 15: Sample output from the automated graph generated by gnuplot (throughput – right hand axis, response time and load – left hand axis)

For better presentation purposes, the majority of the results in this thesis will be presented similarly as the results from Figure 16. At this point, the actual results from these two figures is irrelevant, however the presentation and the kind of information that is expressed is very important.



	Minimum	Median	Average	Maximum	Standard Deviation
Service Response Time (ms)	8.21	10.45	10.53	29.09	1.3
Throughput (trans/sec)	45	1728	1645	2974	877

Figure 16: Sample output from the manual graph containing the same results (throughput – right hand axis) from Figure 15 plus two other metrics (response time and load)

4 DiPerF Scalability and Validation Study

The scalability of DiPerF was performed by testing each component individually prior to testing the entire framework with a scalable enough service. The basic components tested are: 1) communication components (SSH, TCP, UDP), 2) time server, and 3) data analyzer. The communication protocols and the various components are depicted in Figure 17. We also present some large scale performance measurements to show the scalability of DiPerF as an ensemble. It is interesting to note that DiPerF was used as the primary tool to coordinate the scalability study on each component and on DiPerF itself.

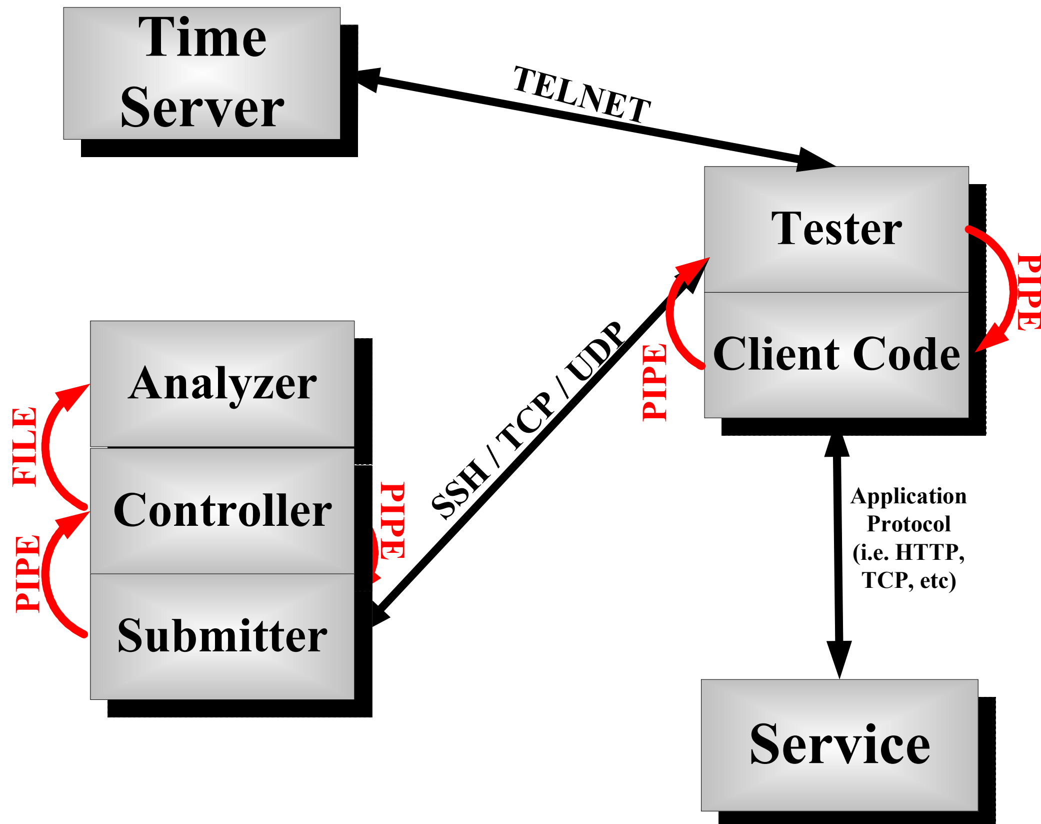


Figure 17: DiPerF Components and Communication Overview

The basic communication component we tested were SSH, TCP, and UDP. In order to show that the testbed supports DiPerF with services of a certain granularity (the finer the granularity, the more traffic is generated), we tested the throughput (in transactions per second) achieved over the various communication protocols. These tests ultimately show the performance of TCP and the performance of the encryption/decryption process for SSH as larger and larger numbers of concurrent connections reached.

The time server component's main function is to answer any incoming connections as quickly as possible with the global time. The time it takes to process a request (including the time the request waits in the wait queue) will ultimately decide the performance of the time server. We used the DiPerF framework to test the time server with a wide range of varying concurrent load, and observed the difference between the service response time and the ping time. We also observe the throughput in transactions per second that can be achieved with the time server.

Finally, to witness the true potential of the DiPerF framework, we decided to test a very lightweight client ("echo") and placed all the various components in motion to see DiPerF in its entirety reach its capacity.

The data analysis is currently implemented as an off-line process, but we test it to evaluate the speed at which it can summarize the collected data.

4.1 Communication Performance

Figure 18 and Figure 19 depict the performance and scalability of the three basic communication methods: TCP, UDP and SSH. For SSH, we investigated two alternatives: SSH RAW which simply wrote all transactions to a file for later analysis, while SSH performed some basic analysis in real-time as it collected all the metrics, and kept all metrics in memory for later easier later analysis. It should be noted that the x-axis and y-axis are depicted in log scale to better visualize the wide range of performance between the various communication methods.

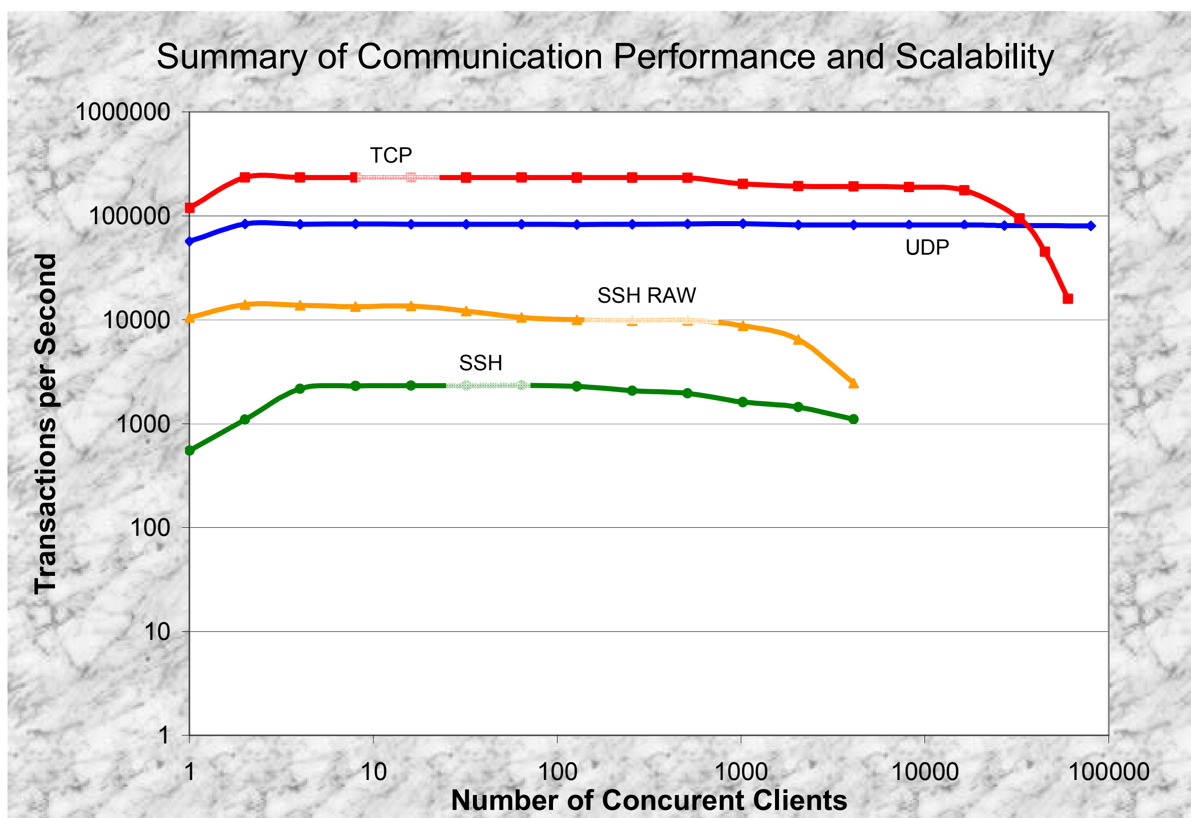


Figure 18: Summary of Communication Performance and Scalability: UDP, TCP, SSH RAW, and SSH

Note that TCP has the best performance for almost all cases except for very large number of concurrent clients (more than 30K clients). At its peak, TCP could process over 230K transactions per second, while at the lowest point (60K concurrent clients), it could still process 15K transactions per second. It is interesting that we were able to reach 60K clients especially that port numbers for TCP are 8 bit values, which means that the most concurrent clients running over TCP we could ever have is fewer than 64K. UDP's performance is somewhat lower than TCP's, but it is much more consistent irrespective of the number of concurrent clients. Overall, UDP achieved about 80K transactions per second up to the 80K concurrent clients we tested. Because the UDP implementation did not provide a reliable communication protocol, the amount of processing that had to be done per transaction increased in order to keep track of the number of messages lost and the number of concurrent machines. This increase in processing decreased the performance of UDP considerably when compared to TCP. The communication based on the SSH protocol was able to achieve 15K transactions per second without any analysis in real time and over 2K transactions per second with some basic analysis. Finally, note the increase in performance from 1 to 2 concurrent clients across all communication protocols. This is owed to the fact that a

dual processor machine was used in handling the multiple client requests; therefore, at least 2 concurrent clients were needed to reach the full capacity of both processors.

Figure 19 shows the performance of each communication method with both extremes (the fewest number of clients, and the most number of clients). Both SSH-based implementations were able to achieve a scalability of 4,000 concurrent clients with a processing overhead of 57 µs and 400 µs for SSH RAW and SSH respectively. On the other hand, TCP achieved 60,000 clients with a processing overhead of 3 µs per transaction. Similarly, UDP achieved 80,000 concurrent clients with a processing overhead of 5 µs per transaction.

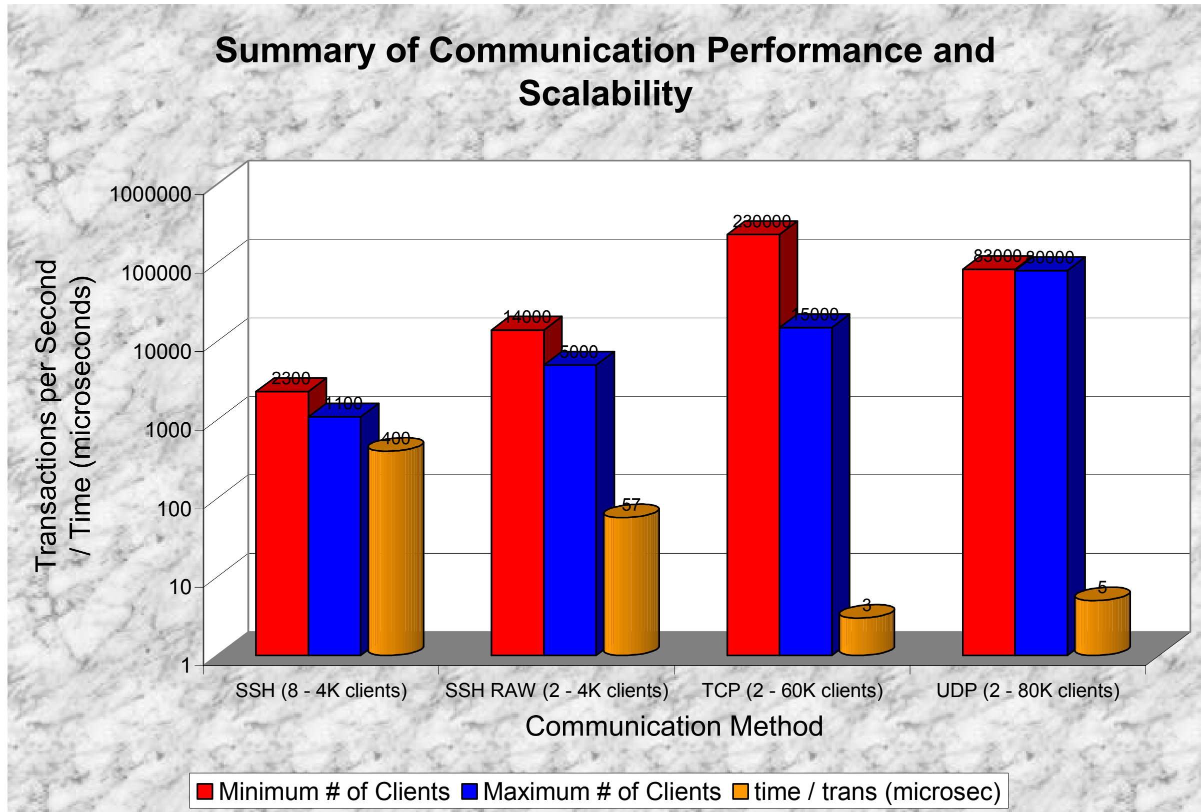


Figure 19: Summary of Communication Performance and Scalability

4.2 Time Server Performance via TELNET

Time synchronization is an important component of DiPerF, and hence the performance of the time server used for synchronizing that time between the controller and the testers is important. The time server uses TELNET as its main communication protocol to reply to time queries. It is interesting to note in Figure 20 that the service response time remained relatively constant (a little over 200 ms) throughout the entire experiment. A total of 1000 clients were used for a total of about 2000 queries per second at its peak. Notice the network latency that it is nearly half the time of the service response time; the difference (about 100 ms) can be attributed to the time it takes for a network packet to traverse the network protocol stack, instantiate a TELNET session, which inherently creates a TCP connection via a 3-way handshake, transfers a small amount of information, and tears down the TCP connection. The majority of the extra time is spent in the 3-way handshake as it requires two network round trip times to establish the TCP connection.

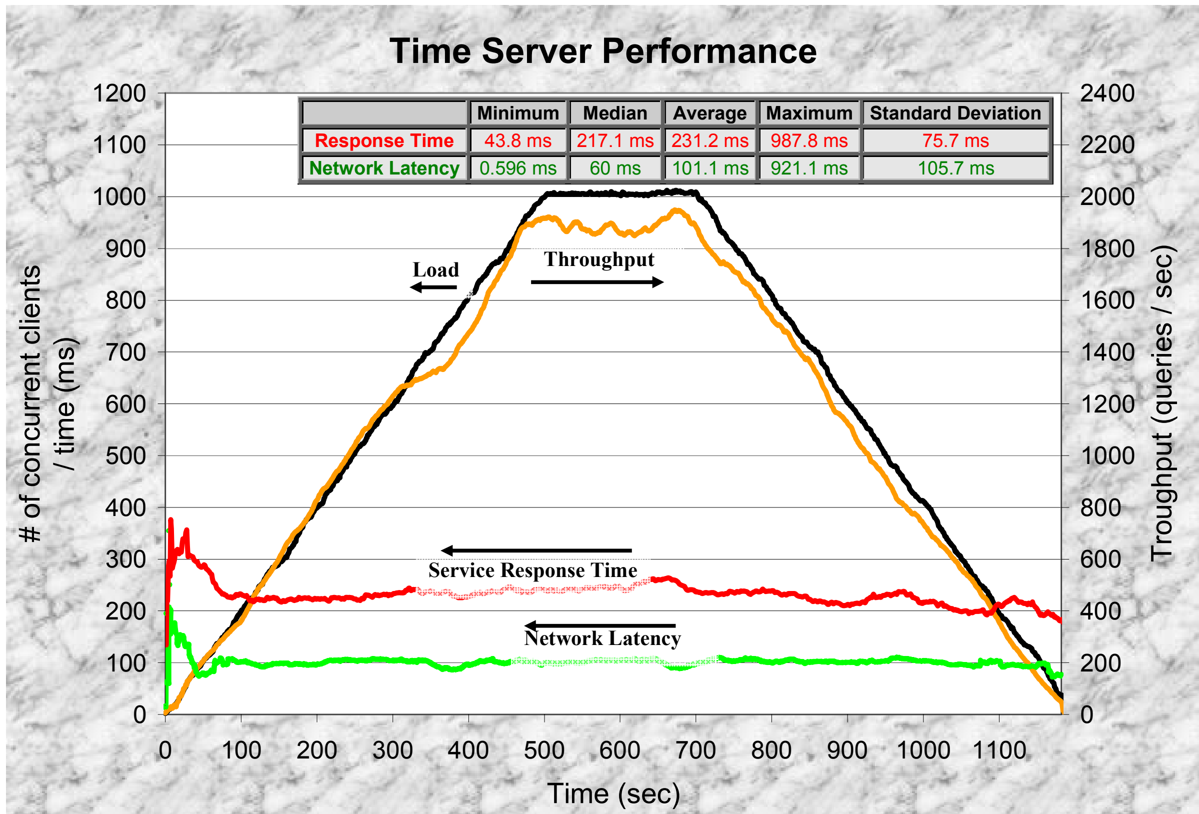


Figure 20: Time server performance with 1000 clients

Our implementation is based on a simple TCP server that accepts incoming connections via "telnet" to request the "global" time. We are currently working on implementing a proprietary client/server model based on the UDP/IP protocol in order to make the time server lighter weight, more scalable, and reduce the response times the clients observe in half by avoiding the 3-way handshake that our current implementation based on TCP has.

4.3 Analyzer Performance

The analyzer component is currently implemented to be an off-line process, and hence the performance of the analyzer is interesting, but not crucial. Because of the many variables that affect the performance of the analyzer, it is hard to generalize the results. Table 3 summarized the performance of the analyzer for five different runs. All the five runs first verified that transaction file that it contained no errors, after which it performed its analysis to extract three metrics: load, throughput, and response time. The main deciding factor in the performance of the analyzer is the number of machines and the number of transaction that need to be processed. For example, for a relatively small run with 40 clients, nearly 3000 transactions which spanned 1000 seconds, it took only one and a half seconds to both verify that data and extract the three metrics. On the other hand, for a larger run, which involved 3600 clients and 354000 transactions, the total time grew significantly to almost 6 minutes. The observed transactions per second throughput achieved by the analyzer varied from 1000 to as high as 9000 transactions per second.

Table 3: Analyzer Performance

Number of Machines	Test Length	Number of Transactions	Memory Footprint (MB)	Time Quantum	Time (sec)	Time/ Trans (ms)	Trans/sec
40	1000	2900	0.54	1 sec	1.6	0.6	1812.5
200	11000	45000	26	1 sec	5.1	0.1	8840.9
1700	6500	671000	146	1 sec	166.5	0.2	4030.0
3600	12000	354000	505	1 sec	359.0	1.0	986.1

4.4 Overall DiPerF Performance and Scalability

In the previous few sections, we showed the performance of the various individual components. This section covers the overall performance of DiPerF as all the components are placed in motion.

4.4.1 Echo Performance via TCP

In the search of finding a scalable service to test the limits of DiPerF, we found the simplest client that did not require network communication, namely a client that simply echoes back a single character ".". This experiment was carried out over the UChicago CS cluster and the DiPerF cluster using 20 machines connected by either 100Mb/s or 1Gb/s network links. Each client was configured to transmit a transaction per second over the TCP-based communication protocol. Note the near perfect performance up to about 45K clients, in which every client manages to have a transaction delivered. Once about 45K concurrent clients were reached, the throughput started dropping to about 15K transactions per second by 60K concurrent clients. It is interesting to bring up that due to limitations of TCP's port number (an 8 bit value), no matter how powerful the hardware was, this implementation could not support more than 64K concurrent clients.

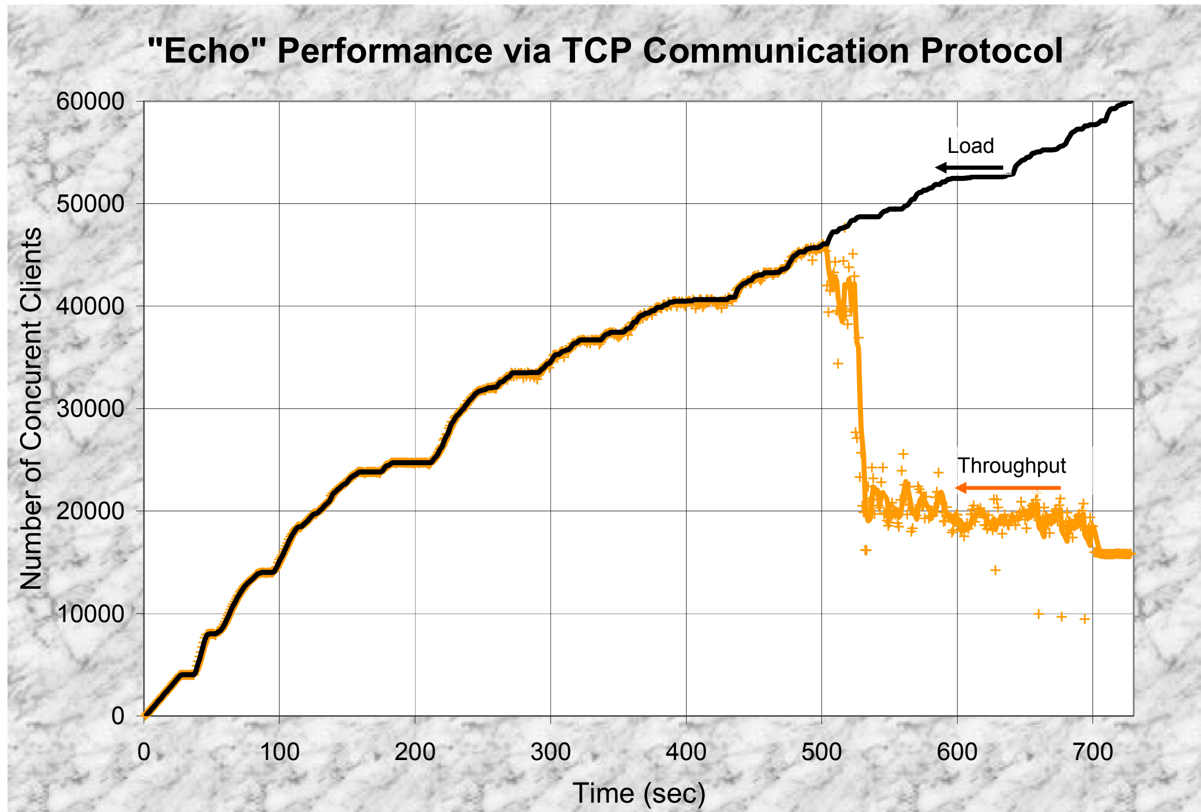


Figure 21: Echo performance via TCP-based Communication Protocol

4.4.2 Echo Performance via UDP

The UDP-based communication protocol implementation performed better than the TCP-based implementation for large number of concurrent clients. It was able to maintain about 1 transaction per second from each client up to 80K concurrent clients. Since UDP does not offer reliability of the transmitted data, we measured the percentage of transactions lost throughout the experiment. we found that as long as the aggregate throughout did not exceed 80K transactions per second, the loss rate was between 1% and 2%.

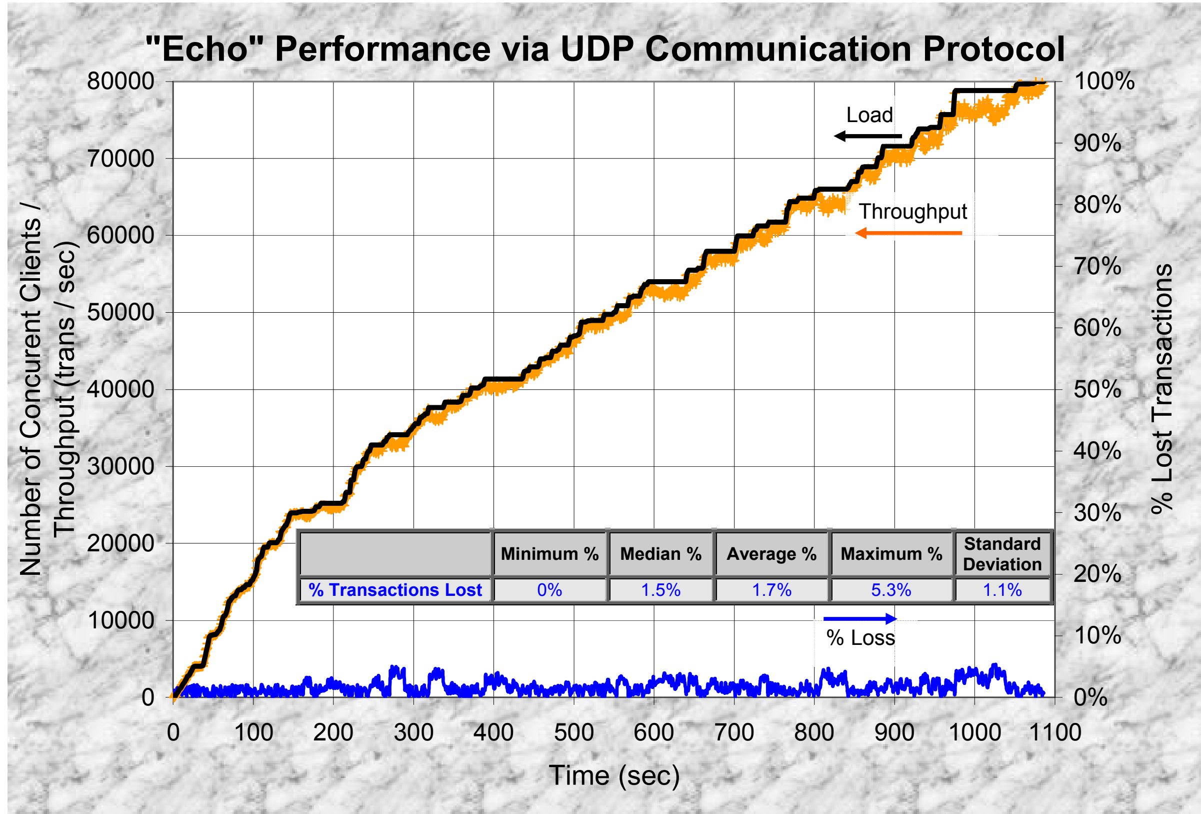


Figure 22: Echo performance via UDP Communication Protocol

4.4.3 Echo Performance via SSH

The performance of the SSH-based communication is clearly lower in both attained throughput and scalability. The main limitation of this implementation is the fact that each remote client instantiates a separate SSH connection, which inherently runs in an isolated process. Therefore, for each new client, there is a new process starting at the controller. Furthermore, each process has a certain memory footprint, plus each TCP connection has the send and receive buffers set to the default value (in our case, 85KB). The memory size can quickly become a bottleneck when there are thousands of processes and TCP connections running even on systems having 1GB of RAM or more. In running this experiment with 4000 clients, DiPerF used the entire 1GB of RAM and the entire 1GB of swap allocated. When DiPerF operates solely out of memory, it is able to achieve over 2000 transactions per second, while at the peak with 4000 concurrent clients, it can only achieve about 1000 transactions per second. Figure 23 shows the performance of the echo client set to run once every 4 seconds at each client.

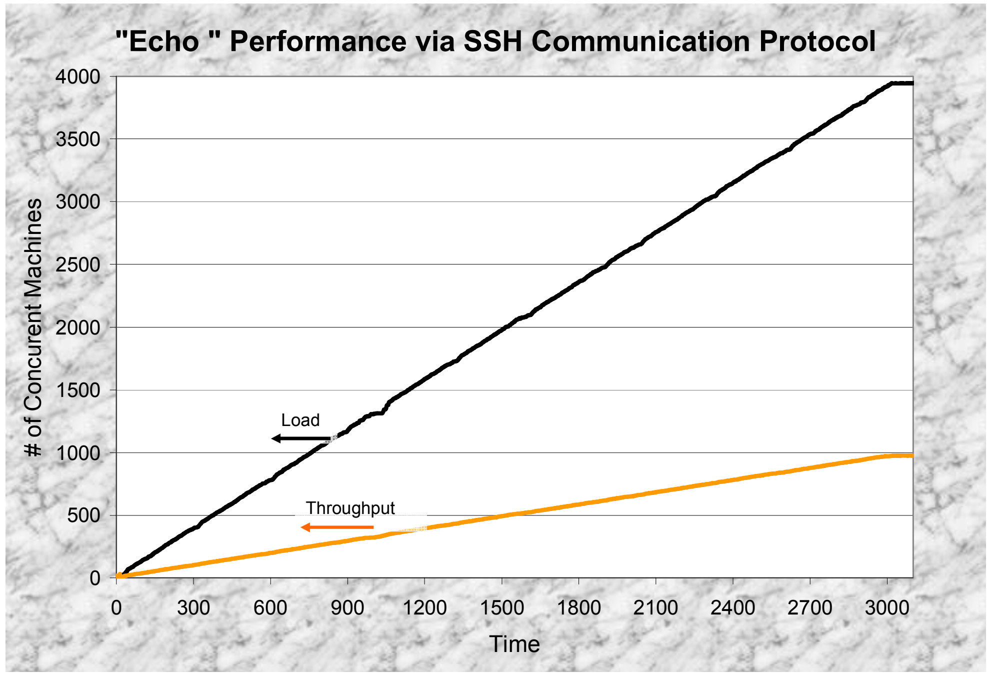


Figure 23: Echo performance via SSH Communication Protocol

4.4.4 GridFTP: real service scaling to 1,000s of clients via SSH

Figure 24 shows the performance of DiPerF when ran over PlanetLab testing the GridFTP server which could scale to 1000s of clients; this experiment used 100+ machines connected via 10Mb/s network links. The throughput showed in Figure 24 represents the achieved network throughput of the GridFTP server in MB/s. A more in-depth analysis of the GridFTP server is offered in a later section.

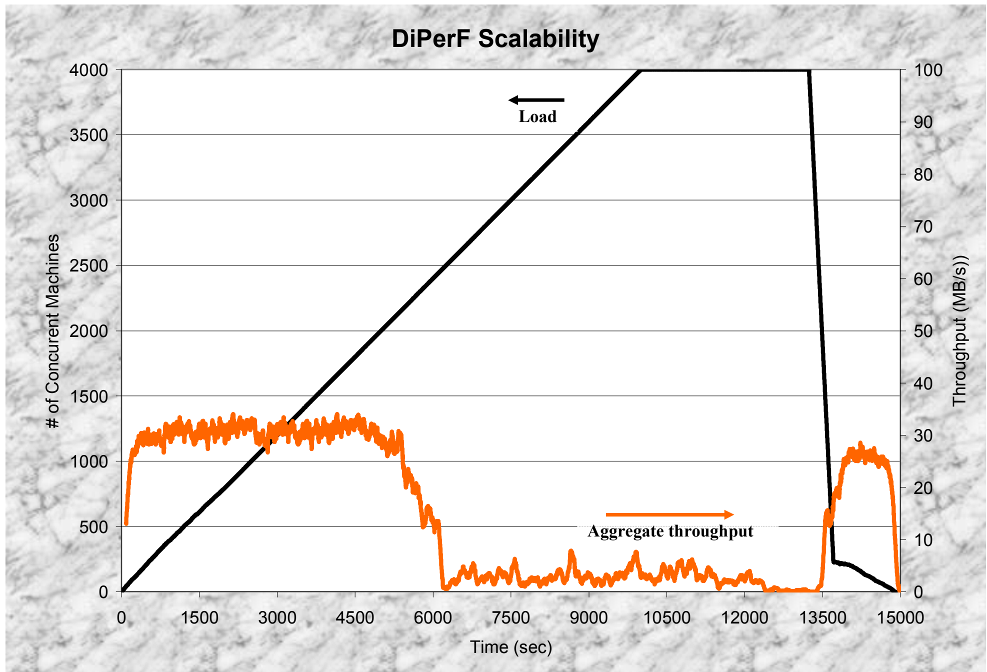


Figure 24: Scalability of DiPerF with 4000 GridFTP clients using SSH

4.5 Validation

This section validates the results produced by DiPerF (client view) via tests performed on the GridFTP server. We compare the aggregate client view as generated by DiPerF with the server view as produced by the Ganglia Monitor [88].

DiPerF generated nearly 80,000 transactions for each metric that generated the client view for the load and for the throughput depicted in Figure 25. The Ganglia Monitor reported the server performance once every 60 seconds, which means that there were about 230 samples generated by Ganglia. That means that there are over 340 DiPerF samples to every Ganglia sample, which could inherently introduce a bigger difference between the two views than there actually exists, especially for fast changing metrics. For example, the load metric changed relatively slow, and was monotonically increasing, which produced about 1% of difference between DiPerF and Ganglia. On the other hand, the throughput observed a higher difference of about 5%, but we believe this is only this large because of the difference sample resolutions between DiPerF and Ganglia. Furthermore, the time synchronization accuracy of about 100 ms could also slightly affect the accuracy of metrics that change very often.

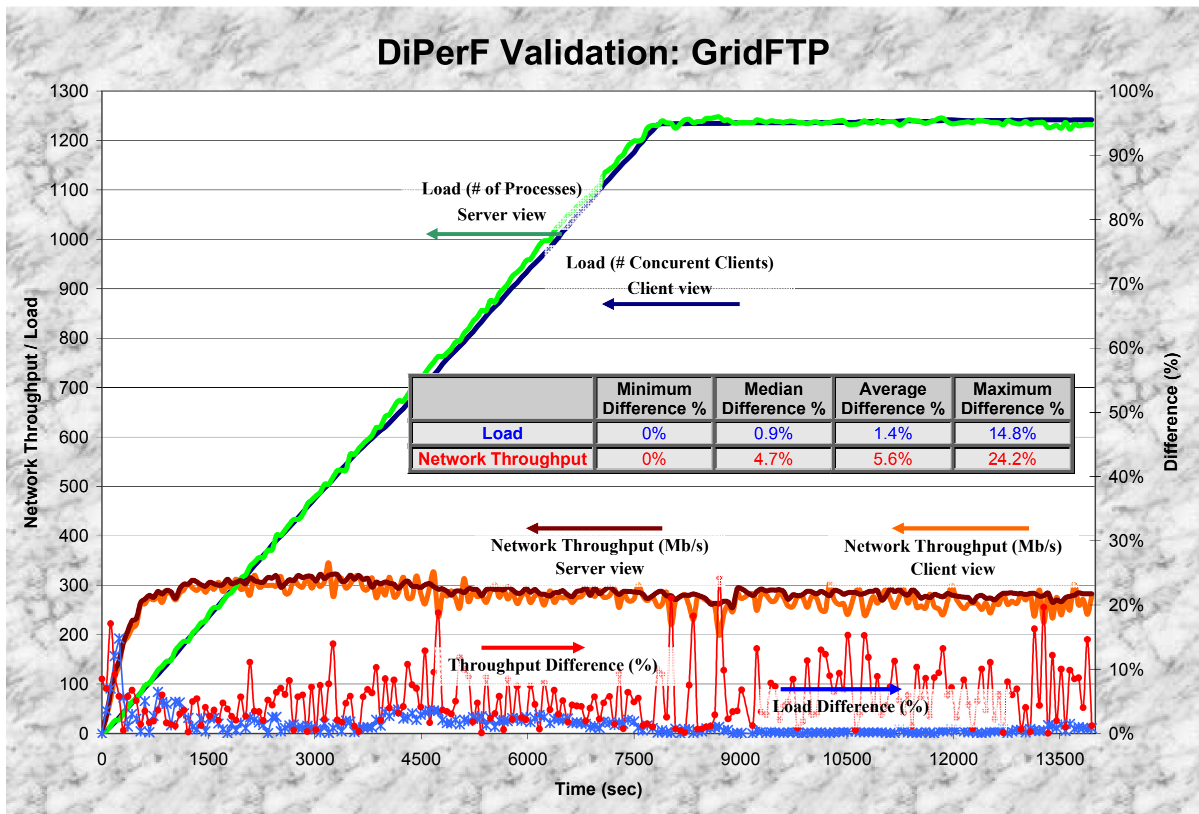
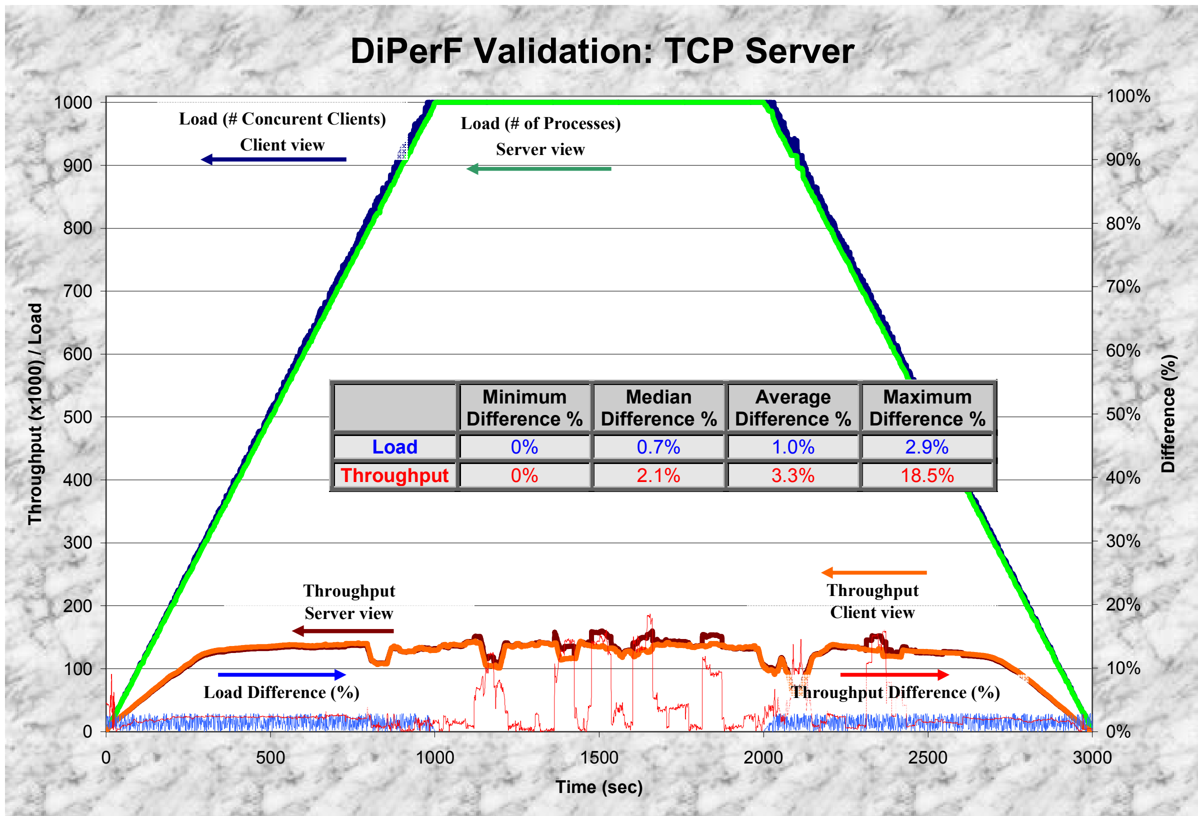


Figure 25: DiPerF Validation against the Ganglia Monitor using GridFTP and 1300 clients in PlanetLab

Figure 25 showed a relatively bad performance of DiPerF between the client and the server view due to the vastly different rates of metric collection. On the other hand, Figure 26 shows a better scenario where both the server and clients all generated samples once a second. DiPerF generated nearly 2,000,000 transactions for each metric that generated the client view for the load and for the throughput depicted in Figure 26. The server monitor reported the server performance once every second; due to there being 1000 concurrent clients, this still generated nearly 1000 client view samples for every server view sample. Due to a finer granularity of samples from the server side, the results are better than those in Figure 25; for the load, we obtained less than 1% difference on average, while for the throughput, we obtained about 3% difference on average.



	Minimum Difference %	Median Difference %	Average Difference %	Maximum Difference %
Load	0%	0.7%	1.0%	2.9%
Throughput	0%	2.1%	3.3%	18.5%

Figure 26: DiPerF Validation against a TCP server using a TCP client and 1000 clients in PlanetLab

Nevertheless, Figure 25 and Figure 26 show that large scale distributed testing aggregate client view can match the server's central view quite well with only a few percent metric value discrepancies on average.

5 Experimental Results

This section covers the experimental results obtained while studying various components of the Globus Toolkit (GRAM, WS-MDS, and GridFTP), and two grid services (DI-GRUBER, and a simple service that performed instance creation). For details on the specific services tested, please see section 2.3.

5.1 GRAM in GT3.2 & GT4

We ran our experiments with 89 client machines for pre-WS GRAM and 26 machines for WS GRAM, distributed over the PlanetLab testbed and the University of Chicago CS cluster (UofC). We ran the target services on an AMD K7 2.16GHz and the controller on an Intel PIII 600 MHz, both located at UofC. These machines are connected through 100Mbps Ethernet LANs to the Internet and the network traffic our tests generates is far from saturating the network links.

The actual configuration the controller passes to the testers is: testers start at 25s intervals and run for one hour during which they start clients at 1s intervals (or as soon as the last client completed its job if the time the client execution takes more than 1s). The client start interval is a tunable parameter, and is set based on the granularity of the service tested. In our case, since both services (pre-WS GRAM and WS GRAM) quickly rose to service response time of greater than 1s, for the majority of the experiments, testers were starting back-to-back clients. Experiments ran for a total of 5800s and 4200s for pre-WS GRAM and WS GRAM respectively. (The difference in total execution time comes from the different number of testers used). Testers synchronize their time every five minutes. The time-stamp server is another UofC computer.

For pre-WS GRAM, the tester input is a standalone executable that was run directly by the tester, while for the WS pre-WS GRAM, the input is a jar file and we assume that Java is installed on all testing machines in our testbed.

5.1.1 GT3.2 pre-WS GRAM

Figure 27, Figure 28, and Figure 29 show results from our pre-WS GRAM job submission experiments. At peak, between seconds 2400 and 3500, all 89 testers run concurrently. Service response time starts out at about 700ms per job, and increases relatively slowly as offered load increases. As the service load surpassed about 33 clients up to 89 concurrent clients (825s to 2225s in the experiment), the service response time, which is already about 7s per job, starts to fluctuate significantly and increase at a faster rate than for the first 33 machines (0s to 825s of the experiment). A similar phenomenon occurs at the end of the experiments when again at a load of about 33 machines, the service response time stabilizes and starts decreasing uniformly. The dashed line depicts a polynomial approximation of the service response time. The solid line depicts a moving average (over a 160 second window) of the response times, which provides a good approximation under normal load (less than 33 concurrent machines) but an increasingly rough approximation as the load approaches the maximum number of nodes of 89.

We conclude that service capacity (or maximum service throughput) for this service deployment is reached with around 33 concurrent clients. The service could probably handle more clients but this would only result in increased response time. It is interesting to note that jobs in sequential processing take about 700ms to run while after running the experiment over the 89 machines over 5780 seconds, 8025 jobs were completed successfully, which nets 720ms per job. The fact that the average time to complete a job remains virtually unchanged when multiple clients concurrently access the service is evidence that each job uses the full capacity of the resources at the service, and hence the service's resource utilization does not increase or decrease with an increasing service load.

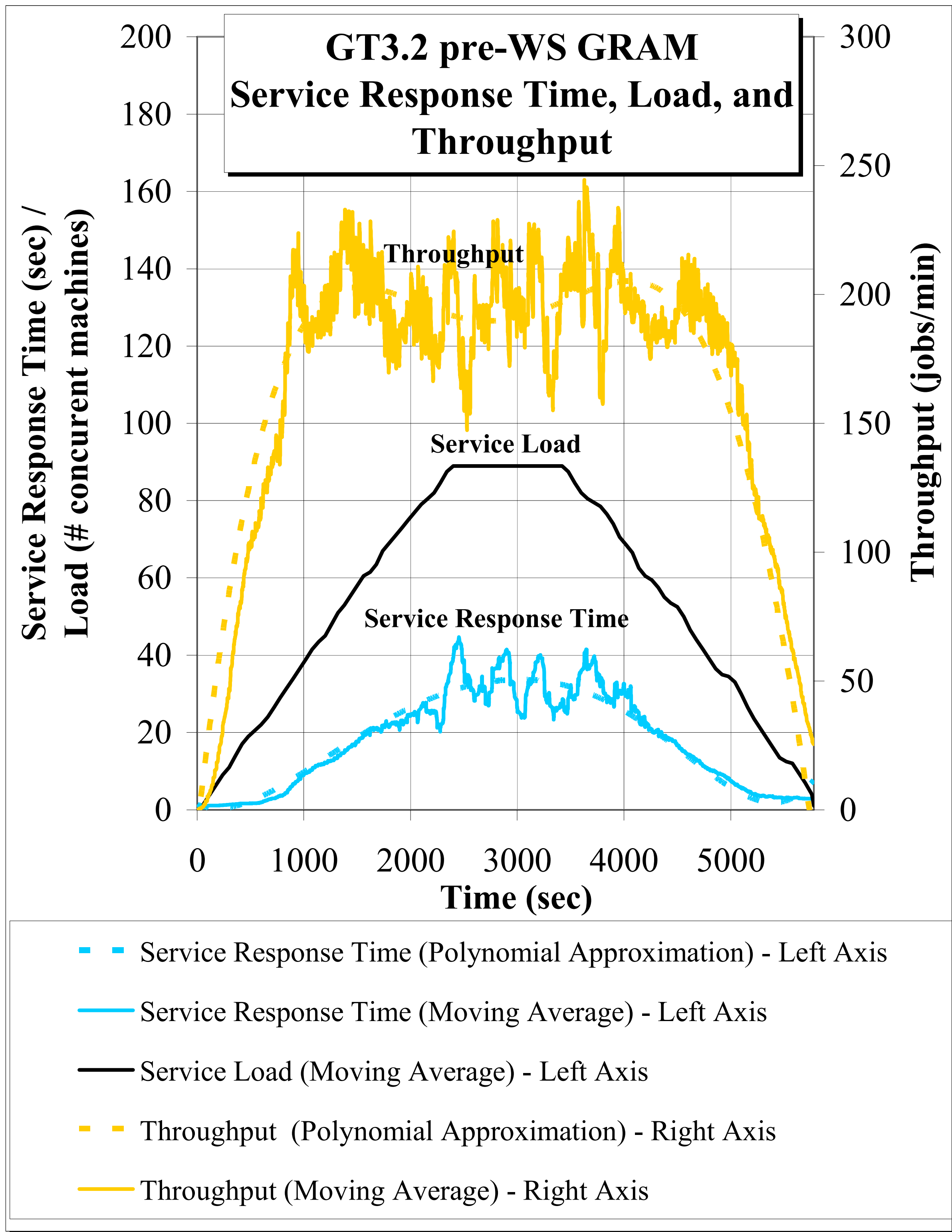


Figure 27: GT3.2 pre-WS GRAM: Service response time, throughput, and service load

Despite a wide range of network latencies observed, service performance remains largely unaffected, which indicates that the time to serve a request is not dominated network latencies. We estimate that the service performance is CPU bound (CPU usage larger than 90% when serving sequential requests). To ensure that creating and destroying the large number of processes the submitted workload involved, was not loading the CPU to unacceptable levels, we verified that we could run the same workload locally (not through the GRAM job submission mechanism) in about 200 seconds. The same workload via job submission took about 5780 seconds, which shows that the actual job execution consumed relatively few resources and the job submission mechanism (GT3.2 pre-WS GRAM) actually consumed the majority of the resources.

Service throughput (measured as the number of requests that complete in a given minute) is the second metric we report. We reiterate that the dashed solid line is a polynomial approximation of the throughput, while the solid line is a moving average approximation. We note that throughput measurements also indicate that throughput peaks at about 33 clients, which supports the claim that the service capacity is reached at 33 concurrent clients. Additionally, increased variability can be observed in the polynomial approximation.

Unlike Figure 27, Figure 28, and Figure 29 present machine (tester) IDs on the *x*-axis. Each machine is assigned a unique ID ranging from 1 to the number of machines used in the experiment; IDs are assigned based on machine relative start time: machine with ID 1 starts first and machine with ID 89 starts last. The values presented on the *y*-axis present a given metric computed for a particular client over the duration of the experiment in which the load was at its peak and all clients were concurrently running; in this experiment, this represents about 1100 seconds. For example, in Figure 28 presents the total service utilization and service fairness per client.

The solid line presents *service fairness*, the ratio between the number of jobs completed and the total service utilization. We see that the service gives a relatively equal share of resources to the clients. Figure 29 also supports the claim that the service gave an equal share of resources to the clients: here, each bubble's surface area denotes the number of jobs completed, which monotonically decreases as the load increases, and monotonically increase as the load decreases. Note that the first few machines (as well as the last few machines) have a lower average aggregate load, which means that they had less competition for the resources, and hence had more jobs completed.

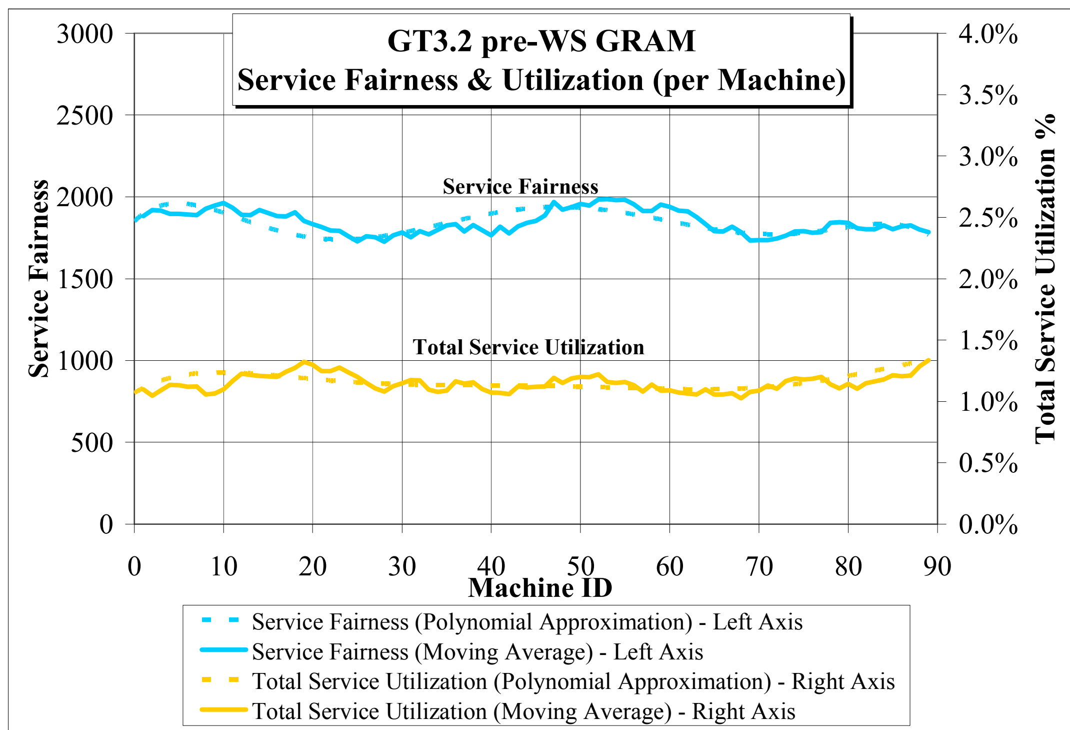


Figure 28: GT3.2 pre-WS GRAM Service utilization per machine

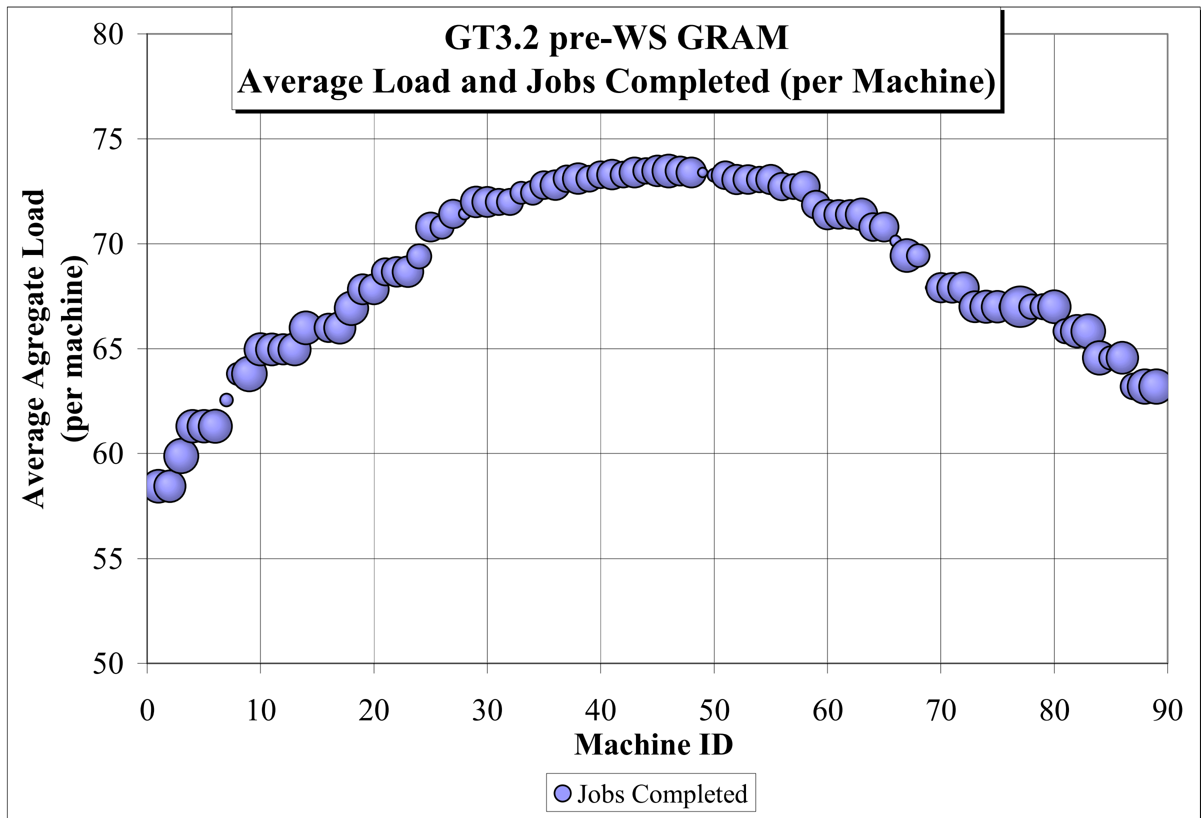


Figure 29: GT3.2 pre-WS GRAM: Average aggregate load vs. number of requests completed per machine. The (x,y) coordinate of the bubble are the machine ID the average aggregate service load respectively; the size of each bubble is proportional to the number of jobs completed by each client

5.1.2 GT3.2 WS GRAM

The next set of experiments presents job submission performance for GT3.2 WS GRAM.

We initially attempted to use the same number of clients (89) as for the pre-WS GRAM tests, but the service did not fail gracefully: after serving a small number of requests, the service stalled and all clients attempting to access the service failed. We ultimately used a pool of 26 machines in testing WS GRAM. The results of Figure 30 show service capacity peaks at around 20 concurrent machines. This result is supported by the fact that the throughput flattens out around the time that 20 machines are accessing the service in parallel, but when the number of machines increases to 26, the throughout decreases dramatically, until a few clients start failing. Once the number of machines decreases to 20 (due to failed clients), the throughout returns to about 10 jobs per minute. Note the service response time decreases as well, once the number of machines stabilized at the service's capacity.

Figure 31 shows that the service fairness varies significantly more than it did for pre-WS GRAM, however based on Figure 32, it seems that only a few clients are not given equal share, which is evident from the few bubbles that have a significantly smaller surface area.

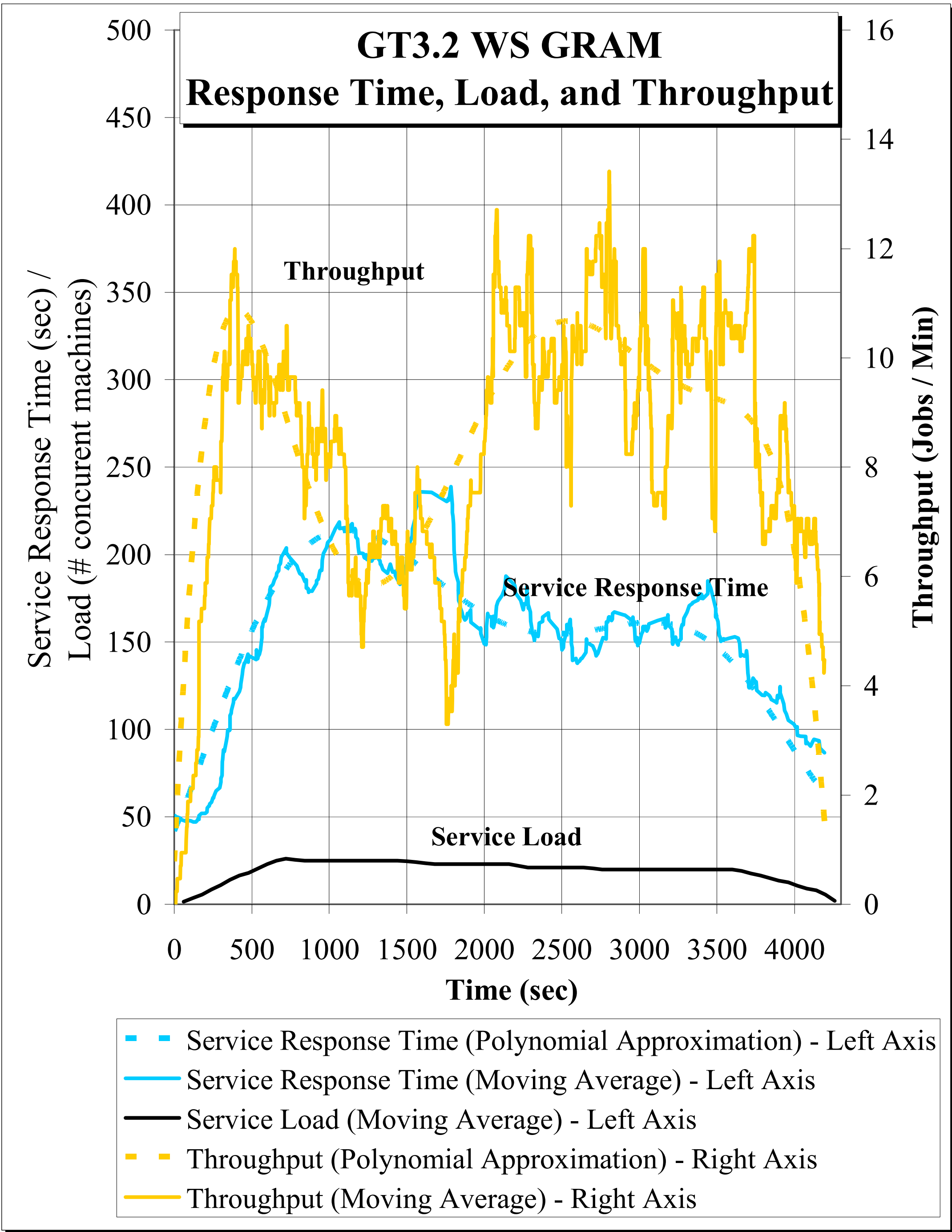


Figure 30: GT3.2 WS GRAM: Response time, throughput, and load

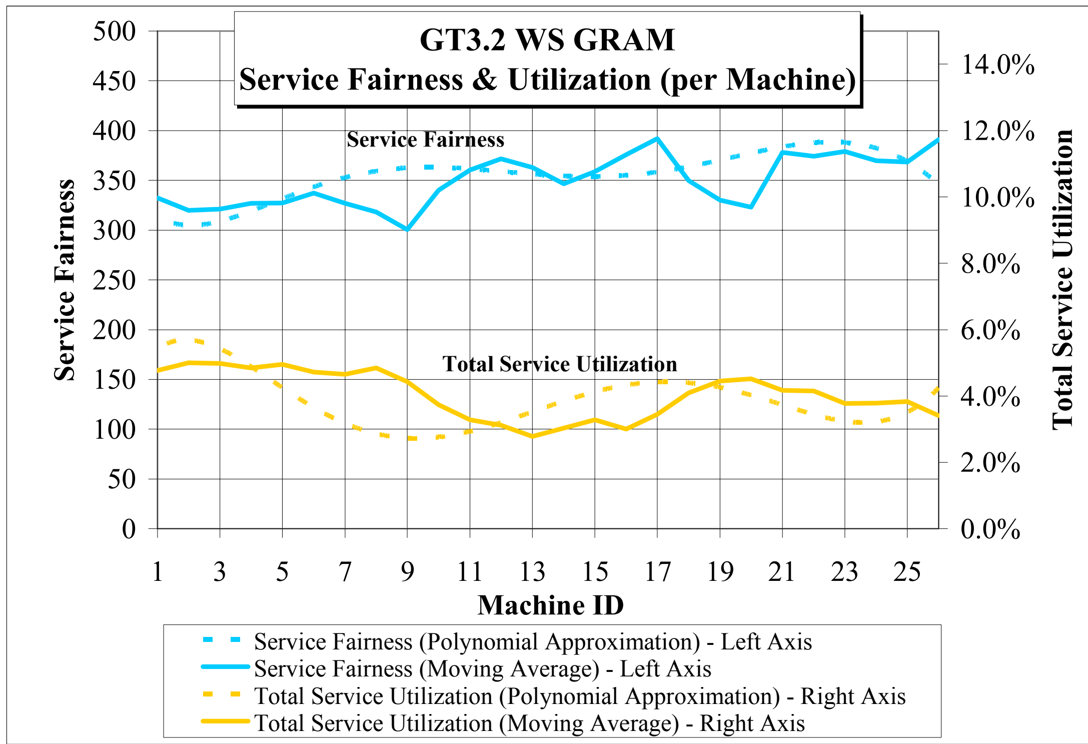


Figure 31: GT3.2 WS GRAM: Service utilization per machine

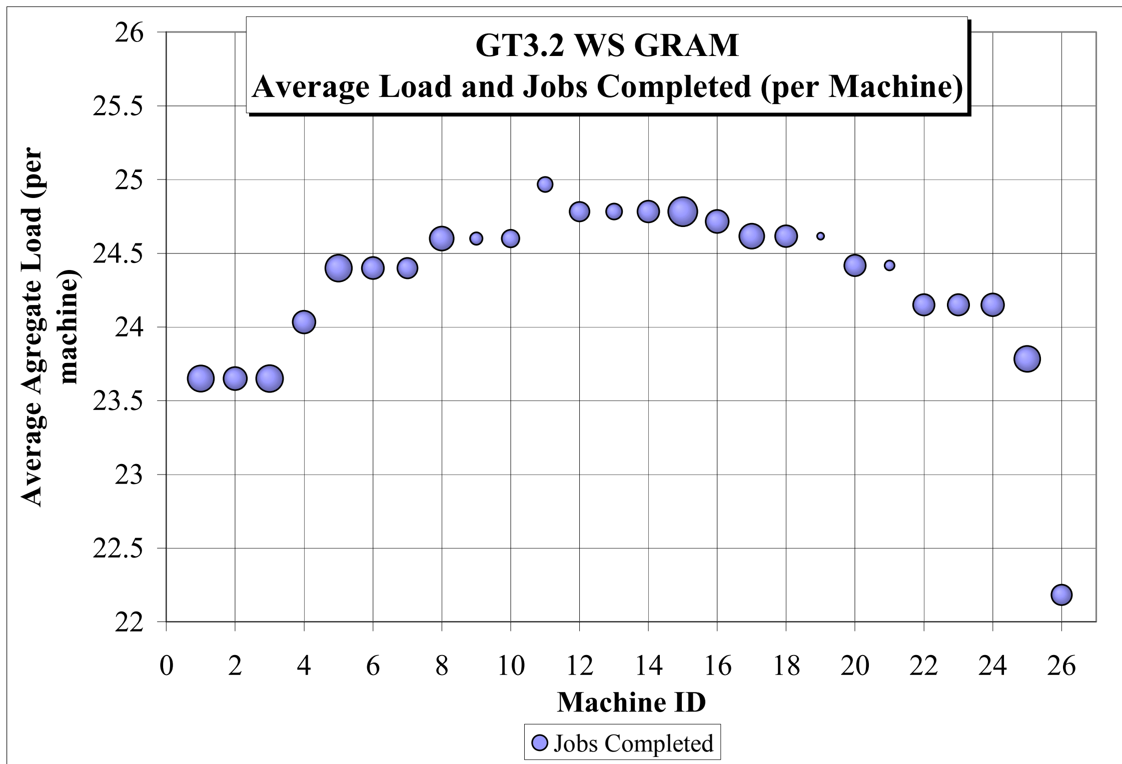


Figure 32: GT3.2 WS GRAM: Average aggregate load and the number of jobs completed per machine; see Figure 29 for the (x,y) coordinates

5.1.3 GT3.9.4 pre-WS GRAM and WS-GRAM Performance Results

We evaluated two implementations of a job submission service bundled with the Globus Toolkit 3.9.4:

- GT3.9.4 pre-WS GRAM (both client and service is implemented in C)
- GT3.9.4 WS GRAM (client is implemented in C while the service is in JAVA)

The metrics collected by DiPerF are:

- *service response time* or time to serve a request, that is, the time from when a client issues a request to when the request is completed minus the network latency and minus the execution time of the client code,
- *service throughput*: number of jobs completed successfully by the service averaged over a short time interval,
- *offered load*: number of concurrent service requests (per second),

We ran our experiments with 115 client machines or less for these experiments; the machines were distributed over the PlanetLab testbed. We ran the target services (GT3.9.4) on an AMD K7 2.16GHz and the controller on an identical machine, both located at UChicago. These machines are connected through 100Mbps Ethernet LANs to the Internet and the network traffic our tests generates is far from saturating the network links.

DiPerF was configured to start the testers at 25s intervals and run each tester for one hour during which they start clients at 1s intervals (or as soon as the last client completed its job if the time the client execution takes more than 1s). The client start interval is a tunable parameter, and is set based on the granularity of the service tested. In our case, since both services (pre-WS GRAM and WS GRAM) quickly rose to service response time of greater than 1s, for the majority of the experiments, testers were starting back-to-back clients.

Experiments ran anywhere from 100 seconds to 6500 seconds depending on how many clients were actually used. Testers synchronize their time every five minutes. The time-stamp server was another UChicago computer.

In the figures below, each series of points representing a particular metric and is also approximated using a moving average over a 60 point interval, where each graphs consists of anywhere from several thousand to several tens of thousands of data points.

5.1.3.1 WS-GRAM Results

Figure 33 depicts the performance of the WS-GRAM C client accessing the WS-GRAM service in JAVA. We note a dramatic improvement from the results of the WS-GRAM service implemented in the Globus Toolkit 3.2 as presented in Figure 30. We observe both greater scalability (from 20 to 69 concurrent clients), and greater performance (from 10 jobs/min to over 60 jobs/minute). We also note that the response time steadily increased with the increased number of clients. The throughput increase seemed to level off at about 15~20 clients, which indicated that the service was saturated and that any more clients would only increase the response time.

Figure 34 shows a very similar experiment as the one showed in Figure 33 which justifies our choice of the number of concurrent clients to use in order to test the WS-GRAM service. Apparently, due to the alpha version of the GT3.9.4 release, there were still new features being added, software "bugs" to be fixed, and performance enhancements to be made. Unfortunately, through our relatively large scale experiments, we managed to trip a scalability problem in which the container would become unresponsive once we reached more than 70 concurrent clients. We therefore reran the experiment with only 69 clients, and obtained the results of Figure 33.

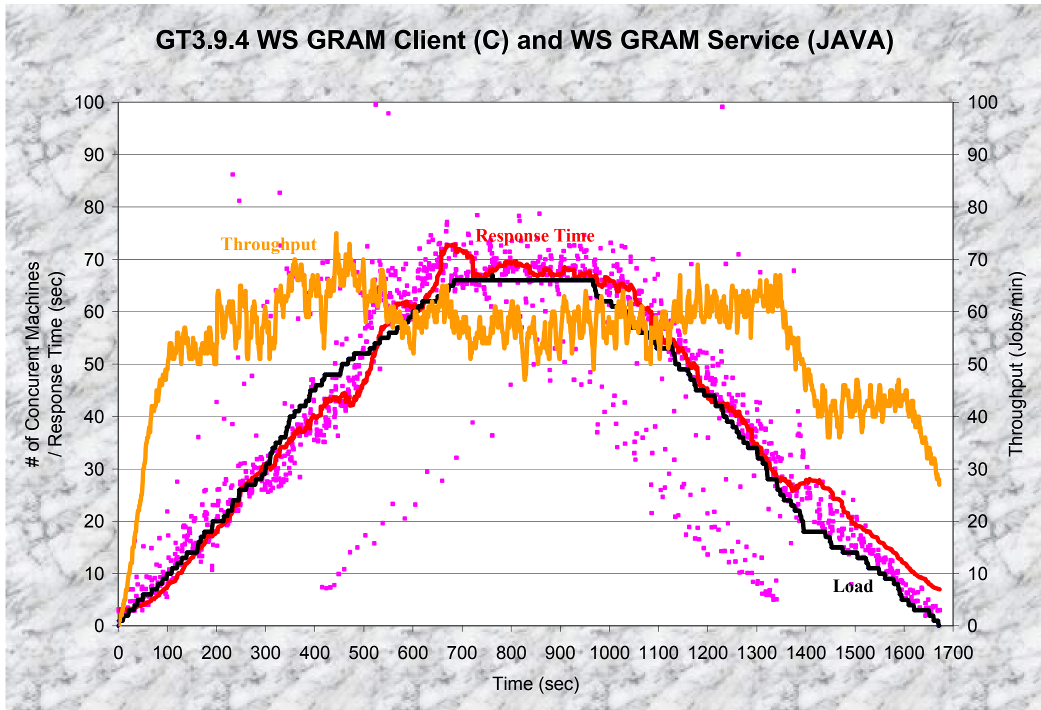


Figure 33: GT3.9.4 WS GRAM client (C implementation) and WS GRAM service (JAVA implementation); tunable parameters: utilized 69 concurrent nodes, starts a new node every 10 seconds, each node runs for 1000 seconds

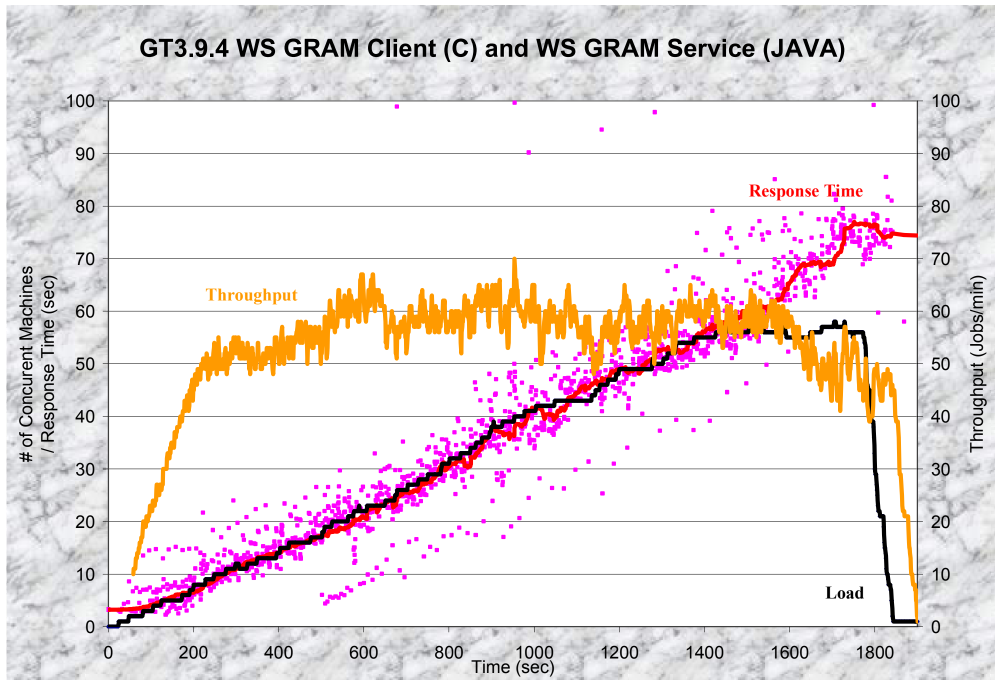


Figure 34: GT3.9.4 WS GRAM client (C implementation) and WS GRAM service (JAVA implementation); tunable parameters: attempted to use 115 concurrent nodes, but after 72 concurrent clients started, the service became unresponsive; a new node was started every 25 seconds, and each node was scheduled to run for 3600 seconds

Figure 35 depicts the performance of the WS-GRAM client with the output enabled. We have enabled the remote client output to be sent back to the originating job submission point, however as can be seen in Figure 35, it seems that this incurs a big performance penalty over the same job submission mechanisms with the output disabled. We attempted to use 69 clients, however, at about concurrent 8 clients, the service became unresponsive.

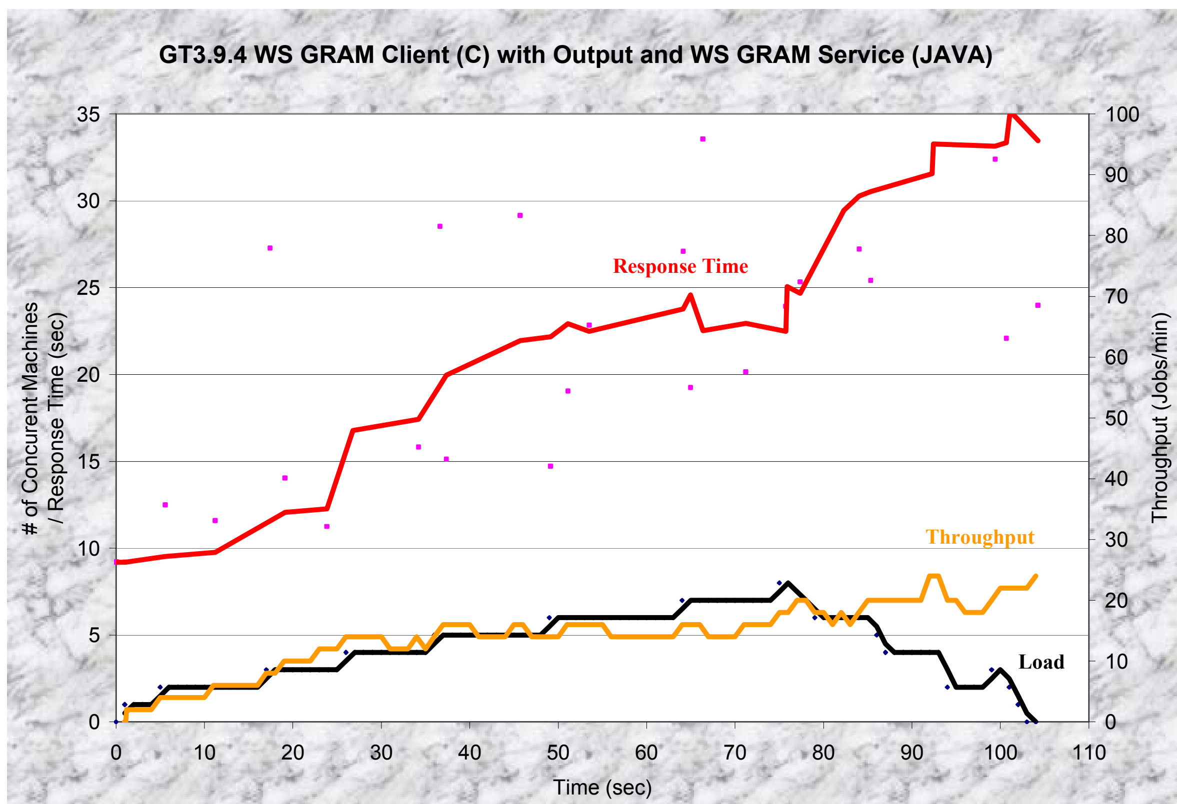


Figure 35: GT3.9.4 WS GRAM client (C implementation) with output enabled and WS GRAM service (JAVA implementation); tunable parameters: 8 concurrent clients, a new node was started every 10 seconds, and each node was scheduled to run for 1000 seconds

5.1.3.2 pre-WS GRAM Results

We performed a similar study on the older pre-WS GRAM implementation that is still bundled with GT3.9.4. We were interested to see how the performance of the pre-WS GRAM compared to that of WS-GRAM in the latest implementation of the Globus Toolkit; furthermore, we were also interested in finding out if enabling the output option had such an adverse effect on the pre-WS GRAM as it did in the WS-GRAM tests.

In comparing the results of the pre-WS GRAM (Figure 36) with those of the WS-GRAM service (Figure 33), we found very similar performance characteristics. The pre-WS GRAM seems to be more scalable, it withstood 115 clients vs. only 69 clients on the WS-GRAM service. Also, the response times seem to be a little less for the pre-WS GRAM service; it achieved service response times in the range of 50~60 seconds for 69 concurrent clients, while the WS-GRAM service achieved response times of close to 70 seconds. The throughout achieve by the pre-WS GRAM service also seems to be more consistent than that achieved by the WS-GRAM service. Based on our results, it seems that pre-WS GRAM with output performs only slightly worse than without output; there is about a 10% performance penalty, which is much lower than what we observed for the WS-GRAM. We see an achieved throughput of slightly less than 80 jobs/min without output and a throughput of slightly more than 70 jobs/min with output. The response times for about 60 clients increased from about 46 seconds to about 51 seconds.

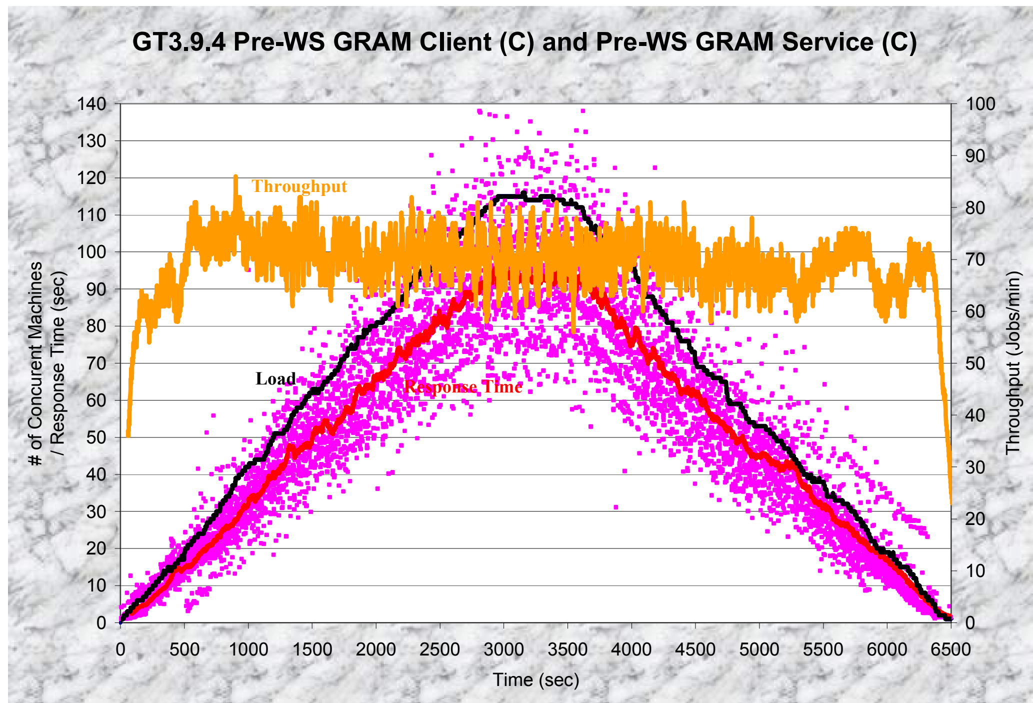


Figure 36: GT3.9.4 pre-WS GRAM client (C implementation) and pre-WS GRAM service (C implementation); tunable parameters: utilized 115 concurrent nodes, starts a new node every 25 seconds, each node runs for 3600 seconds

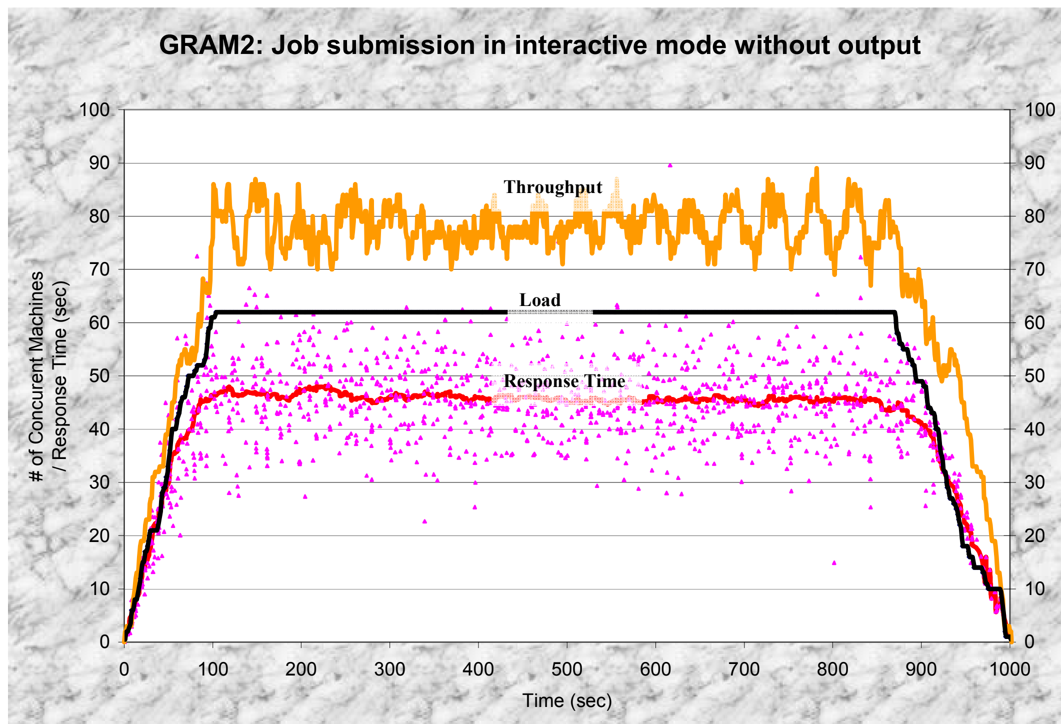


Figure 37: GRAM2 without output

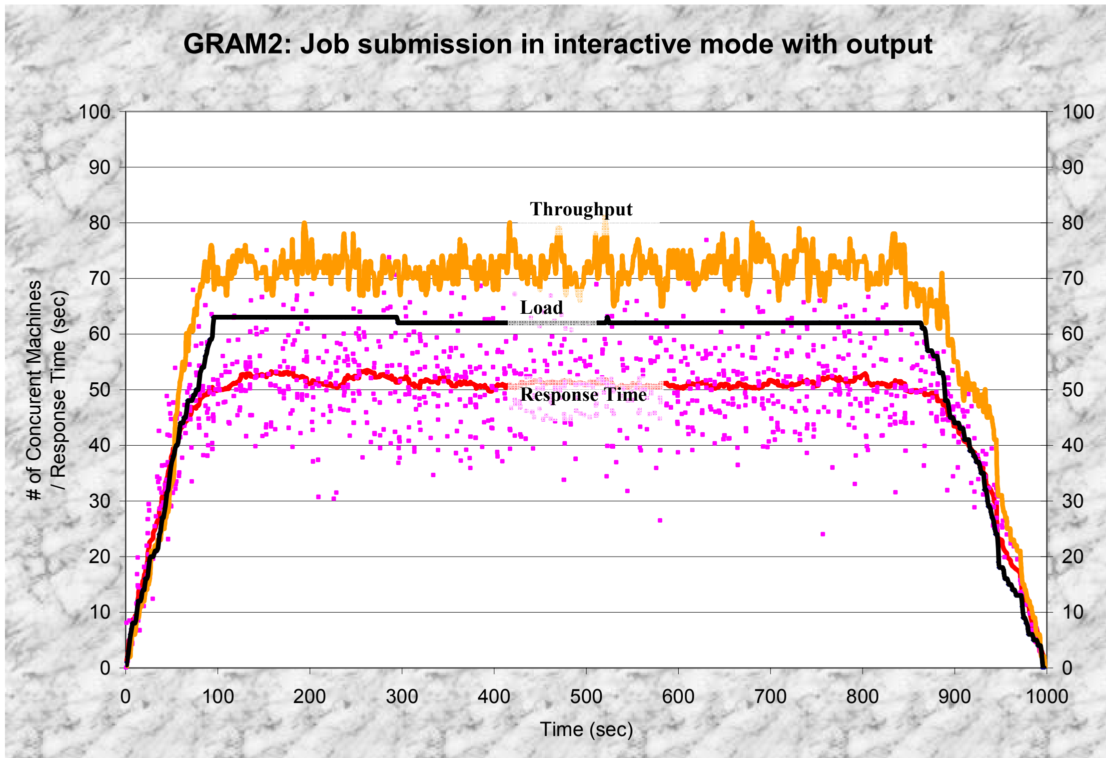


Figure 38: GRAM2 with output

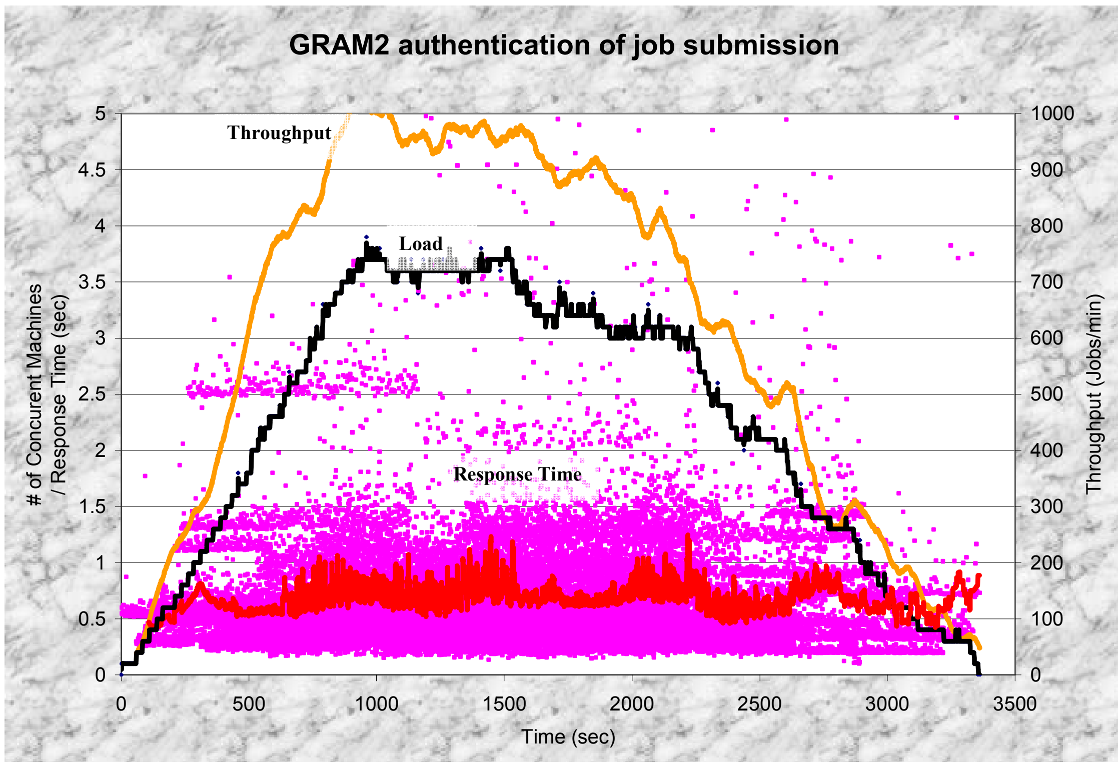


Figure 39: GRAM2 authentication

We also ran some tests to see how long the pre-WS GRAM authentication takes, and we found that it scales very well, at least to up to 38 concurrent clients. We actually had 120 clients in that test, but because of the test parameters (test duration and frequency of clients starting), it ended up that only about 38 clients were ever running concurrently. We see the throughput increase steadily the entire time that more clients join, up to a peak of over 1000 authentications per minute. It is also very interesting to see that the response time stayed relatively constant, between 0.5 seconds and 1 second, regardless of the number of concurrent clients.

5.1.3.3 GT3.9.4 GRAM Conclusions

Comparing the performance of pre-WS and WS GRAM, we find that pre-WS GRAM slightly outperforms WS GRAM by about 10% in both throughput and service response time. When we enabled the remote client output to be sent back to the originating job submission point, the performance of WS GRAM suffers tremendously. Also, for WS GRAM, if we loaded the service with too many concurrent clients, the service became unresponsive. We verified the performance of pre-WS GRAM with output and without output, and we found that the performance loss with output was only about 10%.

5.1.4 GRAM Summary

Table 4 summarizes the performance of pre-WS and WS-GRAM across both GT3 and GT4 found in the previous subsection. The experiments performed on GT3 and GT4 were done almost 1 year apart, and hence the results are not directly comparable between GT3 and GT4, but they should give a rough overview of the performance of both releases and implementation. It is noteworthy to point out the dramatic performance improvement WS-GRAM experienced from GT3 to GT4, going from less than 10 jobs per minute to almost 60 jobs per minute, and getting a reduction of response times from almost 180 seconds to less than 70 seconds.

Table 4: GRAM performance summary covering pre-WS GRAM and WS-GRAM under both GT3 and GT4

Experiment Description	Throughput (transactions/sec)					Load at Service Saturation	Response Time (sec)				
	Min	Med	Aver	Max	Std. Dev.		Min	Med	Aver	Max	Std. Dev.
Figure 27: GT3 pre-WS GRAM – 89 clients	99.3	193	193	326	38.9	33	1.02	20.4	31.5	739	41.7
Figure 30: GT3 WS-GRAM – 26 clients	3.13	9.16	8.77	12.8	2.19	20	35.6	178.4	173.6	298	55.9
Figure 36: GT4 pre-WS GRAM – 115 clients	57	69.8	69.8	81.6	4.23	27	57.6	92	93	167.2	15.1
Figure 33: GT4 WS-GRAM – 69 clients	47.2	57	56.8	63.9	3.1	20	32.1	67.7	67.4	131.8	9.8

5.2 GridFTP

In order to complement the performance study [20] on the GridFTP server done in a LAN, we performed another study in a WAN environment mainly targeting the scalability of the GridFTP server.

The metrics collected (client view) by DiPerF are:

- **Service response time** or time to serve a request, that is, the time from when a client issues a request to when the request is completed minus the network latency and minus the execution time of the client code; each request involved the transfer of a 10MB file from the client hard disk to the server's memory (/dev/null)
- **Service throughput**: aggregate MB/s of data transferred from the client view

- **Load**: number of concurrent service requests

The metrics collected (server view) by Ganglia are, with the *italicized* words being the metric names collected from Ganglia:

- **Number of Processes**: *proc_total* – (number of processes running at start of the experiment)
- **CPU Utilization**: *cpu_user* + *cpu_system*
- **Memory Used**: mem_total + swap_total – mem_free – mem_cache – mem_buffers – mem_share – swap_free
- **Network throughput**: *bytes_in* (converted to MB/s); we were only interested in the inbound network traffic since we were performing uploads from clients to the server

We ran our experiments on about 100 client machines distributed over the PlanetLab testbed throughout the world. Some of the later experiments also included about 30 machines from the CS cluster at University of Chicago (UChicago). We ran the GridFTP server at ISI on a machine with the specs outlined in Table 2 in section 2.3; the DiPerF framework ran on an AMD K7 2.16GHz with 1GB RAM and 100Mb/s network connection located at UofC. The machines in PlanetLab are generally connected by 10Mbps Ethernet while the machines at UChicago are generally connected by 100Mbps Ethernet.

For each set of tests, the caption below the figure will address the particular configuration of the controller which yielded the respective results. We also had the testers synchronize their time every five minutes against our time server running at UChicago. In the figures below, each series of points represents a particular metric and is also approximated using a moving average over a 60 point interval, where each graphs consists of anywhere from 1,000s to 100,000s of data points.

5.2.1 GridFTP Scalability

The results from Figure 40 were taken to establish the base performance giving us an overview of the achievable throughput between the PlanetLab testbed and ISI. It is interesting to note that the network throughput steadily grew with every new concurrent client up to over 30 MB/s. At the same time, the service response time (the time taken to transfer a 10 MB file from 1 client to the server) remained relatively constant with a response time of about 25 seconds. A transfer time of about 25 seconds equates to about 3.2 Mb/s network throughput from each client to the server.

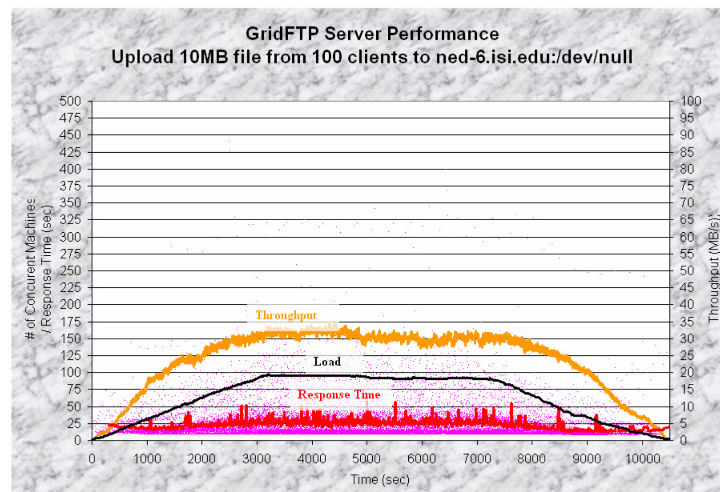


Figure 40: GridFTP server performance with 100 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 100 concurrent clients, starts a new client every 30 seconds, each client runs for 7200 seconds; 243.4 GB of data transferred over 24,925 file transfers; left axis – load, response time; right axis - throughput

In the experiment from Figure 41, we increased the number of clients to 500 on the same set of 100 physical machines. It was interesting to see that running multiple clients on the same host improved the aggregate throughput from around 30 MB/s to almost 40 MB/s; this is evidence that the large network latencies in the WAN environment were limiting the performance of certain nodes that had a high bandwidth-delay product. We notice that in the first 100 clients (1 client per machine), the throughput reached the same 30 MB/s as in Figure 40; by the time the load reached 200 concurrent clients (2 clients per machine), the throughput reached 40 MB/s, after which it remained relatively constant until the load started decreasing. We also notice the response time increasing steadily, as more and more clients are being utilized on the same physical nodes, and hence the multiple clients are splitting their available network bandwidth among each other.

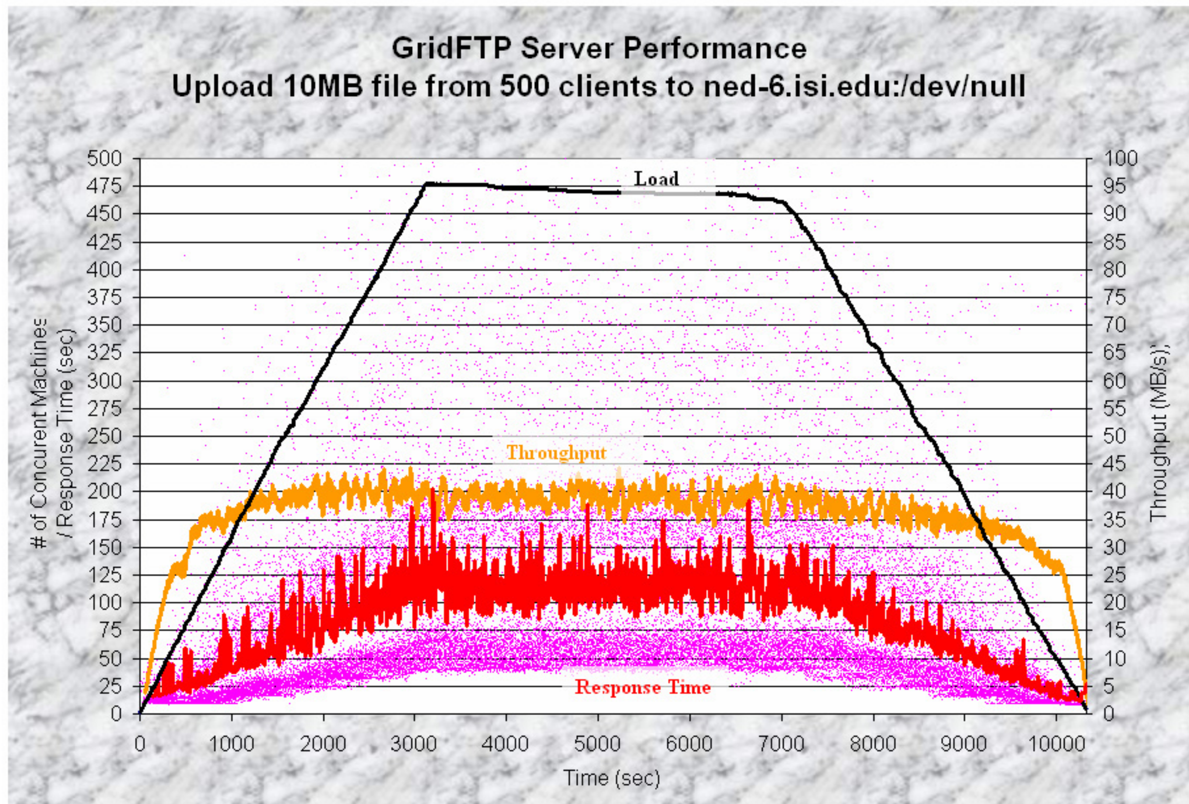


Figure 41: GridFTP server performance with 500 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 500 concurrent clients, starts a new client every 6 seconds, each client runs for 7200 seconds; 363.3 GB of data transferred over 37,201 file transfers; left axis – load, response time; right axis - throughput

The experiment from Figure 42 involved 30 more machines located at UChicago on top of the 100 PlanetLab machines; we also more than doubled the number of clients from 500 to 1100. We notice that the peak aggregate throughput remains the same, around 40 MB/s, despite the fact that independent tests of the PlanetLab testbed yielded around 40 MB/s and similar tests of just the UChicago CS cluster testbed yielded around 25 MB/s. We conclude that since the testbeds could achieve an aggregate throughput of 65 MB/s, the limitation that yielded around 40 MB/s was either the server or the network connection to ISI. The server seemed to have ample CPU resources available; furthermore, after performing some LAN tests at ISI, we found that the server could reach a network throughput of over 100 MB/s, which makes it likely that the connection into ISI was the bottleneck.

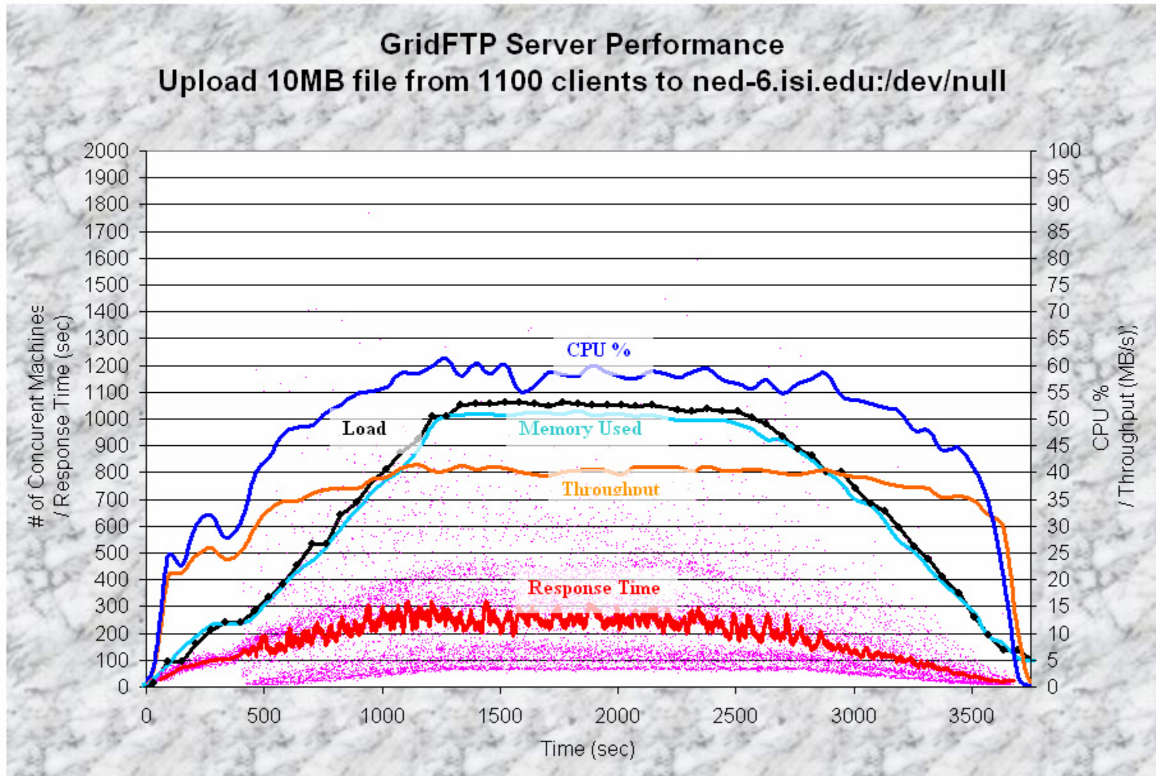


Figure 42: GridFTP server performance with 1100 clients running on 100 physical nodes in PlanetLab and 30 physical nodes in the CS cluster at UofC; tunable parameters: utilized 1100 concurrent clients, starts a new client every 1 second, each client runs for 2400 seconds; 131.1 GB of data transferred over 13,425 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %

When comparing some of the metrics from Ganglia and those of DiPerF, it is interesting to note that the memory used follows pretty tight the number of concurrent clients. In fact, we computed that the server requires about 0.94 MB of memory for each new client it has to maintain state for; we found this to be true for all the tests we performed within +/- 1%. Another interesting observation is that the CPU utilization closely mimics the achieved throughput, and not necessarily the number of concurrent clients. This clean separation between the scalability and performance of the GridFTP server makes it relatively easy to do capacity planning in order to achieve the desired quality of service provided by the server.

Figure 43 is really interesting because it showed the behavior of the server under a changing network condition and under a changing processor load. A little background on PlanetLab and its policies is useful in understanding the results from this test. Once a slice (which could run multiple clients) sends 16GB in a day on a particular node, the slice is then limited to 1.5Mbps for the rest of the day on that node. The bandwidth cap is implemented as a very big 16GB token bucket; a slice is permitted to burst (up to the physical line rate) until it sends 16GB, then the slice gets rate limited to 1.5 Mb/s. Once a day, the token bucket is reset and the whole process starts again.

The results from Figure 43 represent exactly this behavior. We had been running many tests prior to this one, so we probably managed to get the better connected hosts to trigger their 16GB cap, and limit themselves to only 1.5 Mb/s. In the middle of the tests, apparently there were some nodes that reset their counters, and all of a sudden those nodes had significantly more bandwidth to use, and hence we see the increase from 30 to 40 MB/s and a CPU utilization from 45% to 65%. After another hour of the experiment, the CPU started being utilized 100% due to a competing process outside of our experiments. In order to put things into perspective for this experiment in terms of the amounts of data we transmitted in this experiment, we transferred about 916 GB of data over almost 94,000 files transfers.

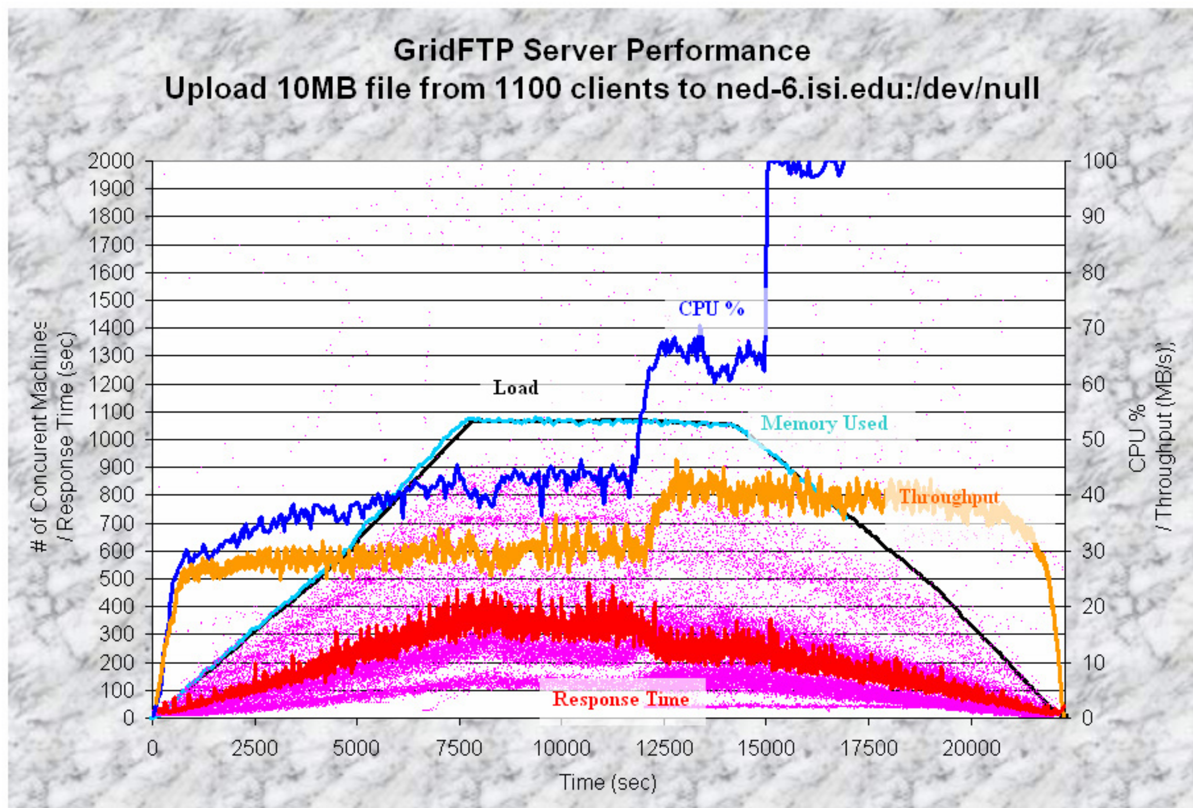


Figure 43: GridFTP server performance with 1100 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 1100 concurrent clients, starts a new client every 6 seconds, each client runs for 14400 seconds; 916.58 GB of data transferred over 93,858 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %

Figure 44 is really interesting because it tries to depict both the client and server view simultaneously in order to validate the results from DiPerF. The two metrics that line up nearly perfect are: 1) load (number of concurrent clients {black} vs. number of processes {gray}) and 2) throughput (aggregate client side {light orange} vs. server side {dark orange}). If it is hard to discern between the client and server view, that is because they light up almost perfectly most of the time, and it is only in certain small places where they diverge enough to see that the data contains two separate metrics. Notice how the achieved throughput is steadily decreasing, as we are probably hitting the 16GB per day limits on certain nodes, and hence we get a lower aggregate throughput by the end of the experiment.

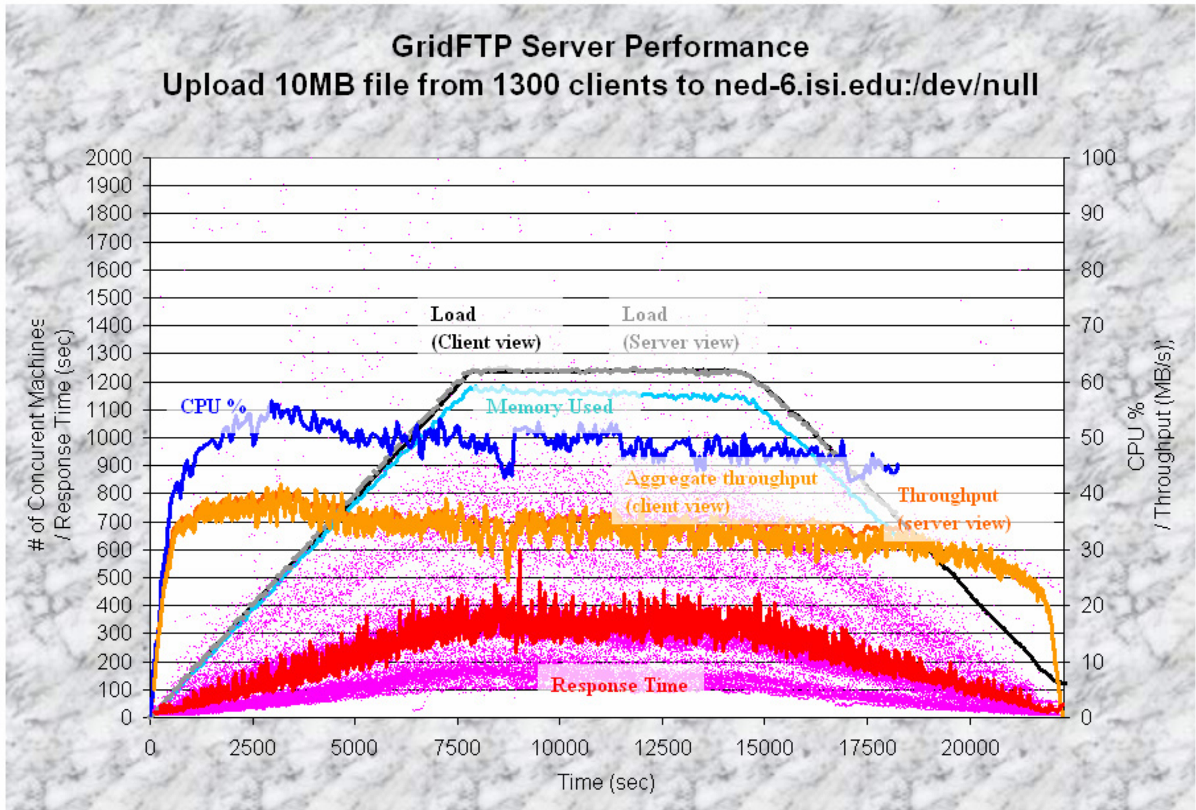


Figure 44: GridFTP server performance with 1300 clients running on 100 physical nodes in PlanetLab; tunable parameters: utilized 1300 concurrent clients, starts a new client every 6 seconds, each client runs for 14400 seconds; 767 GB of data transferred over 78,541 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %

Figure 45 is probably the most impressive due to the scale of the experiment. We coordinated 1800 clients over 130 physical nodes distributed around the world. It is very interesting to see that the throughput reached around 45 MB/s (~360 Mb/s) and stayed consistent throughout despite the fact that the server ran out of physical memory and started swapping memory out; with the OS footprint in the neighborhood of 100 MB, and the 1700 MB of RAM used by the GridFTP server to serve the 1800 concurrent clients, the system ended up using about 300 MB of swap and the entire 1.5 GB of RAM. Note that the CPU utilization is getting high, but with a 75% utilization and another 1.5GB of swap left, the server seems as if it could handle additional clients. From a memory point of view, we believe that it would take about 5000 concurrent clients to leave the server without any memory or swap in order to cause it not be able to handle new incoming clients. From a CPU point of view, we are not sure how many more clients it could support since as long as the throughput does not increase, it is likely that the CPU will not get utilized significantly more. Another issue at this scale of tests is the fact that most OSes have hard limits set in regards to file descriptors and number of processes that are allowed to run at any given point in time. With 1800 clients, the experiment saturated the network link into the server, but it is unlikely that the server's raw (CPU, memory, etc…) resources were saturated.

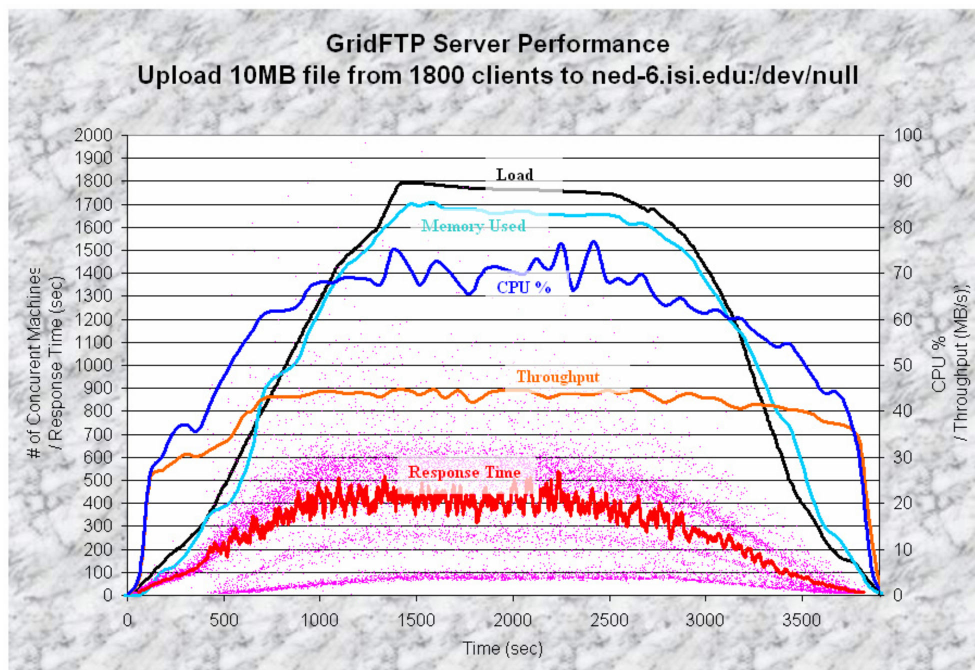


Figure 45: GridFTP server performance with 1800 clients running on 100 physical nodes in PlanetLab and 30 physical nodes in the CS cluster at UofC; tunable parameters: utilized 1800 concurrent clients, starts a new client every 1 second, each client runs for 2400 seconds; 150.7 GB of data transferred over 15,428 file transfers; left axis – load, response time, memory; right axis – throughput, CPU %

5.2.2 GridFTP Fairness

In order to quantify the fairness of the GridFTP server, we investigated the fairness at two levels. The first level was the fairness between the different machines (about 100 nodes in PlanetLab located in the USA) in relation to their network connectivity to ISI. The second level was the fairness among the different clients running on the same physical nodes, essentially comparing the fairness among competing flows on the same machine. To make the results the most meaningful, the results presented in this section were a subset of an entire experiment in which all the clients were running in parallel; in other words, we omitted the data when the number of clients were increasing and decreasing, and just kept the peak.

Figure 46 shows the amount of data transmitted from each machine which ran 15 clients each. The x-axis represents the machine's network latency to ISI, and the y-axis represents the machine's network bandwidth as measured by IPERF. The size of the bubble represents the amount of data transmitted by each machine

throughout the 3 hour experiment; the largest bubbles represent a transfer amount of over 7 GB, while the smallest bubble represents a transfer of just 0.5 GB. It is interesting to note that network latency plays a significant role in the network bandwidth, and essentially in the achieved performance of the GridFTP client. Based on the results Figure 46, we claim that the GridFTP server gave a fair share of resources to each machine according to the machines network connectivity.

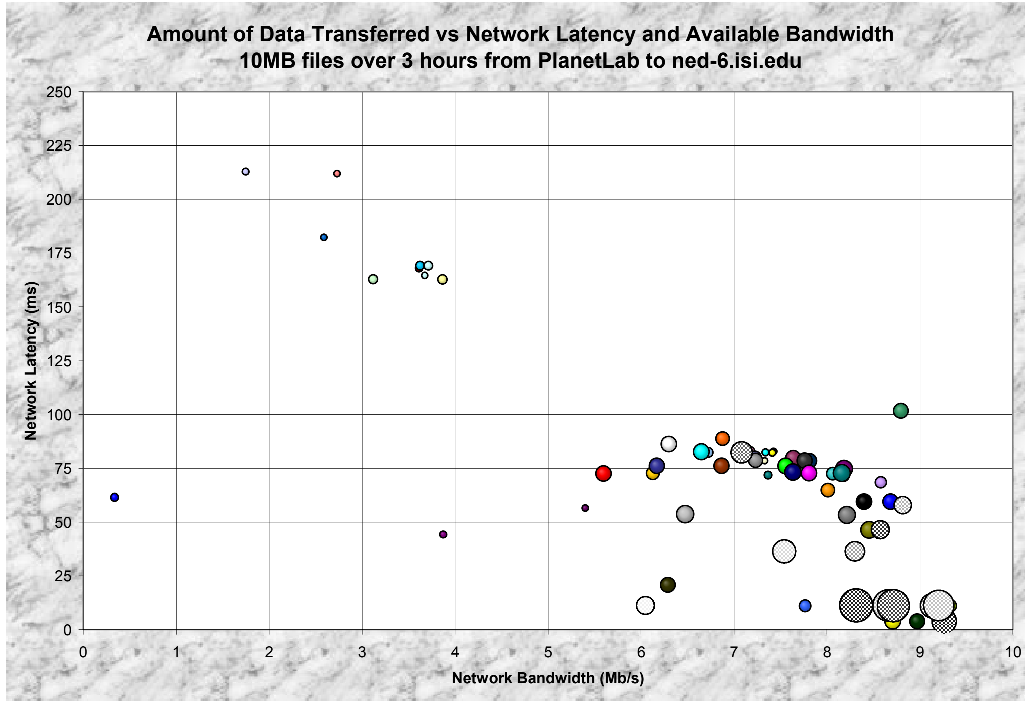


Figure 46: GridFTP resource usage per machine using 10MB files

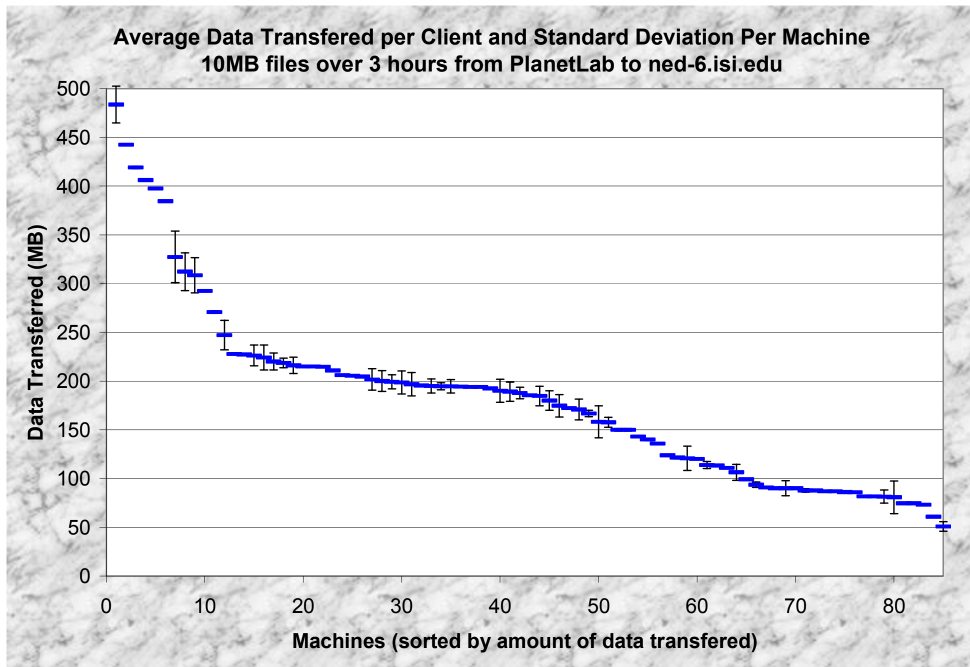


Figure 47: GridFTP resource usage per client using 10MB files

On the other hand, Figure 47 attempts to capture fairness between various competing flows running on the same physical machines. The machines have been ordered with the best connected machines first and the worst connected machines last in order to make the figure more readable. The blue horizontal line represents the average data transferred per client of a particular machine, while the black vertical line represents the standard deviation of the results of all clients on a particular machine; in this experiment, we had 15 clients run concurrently on each machine, so the standard deviation is calculated from these 15 values for each machine. It is very interesting to notice that the standard deviation was very small, with only a few percent of deviation of performance from client to client running on the same host.

Using 10 MB files for each transaction (as we did in Figure 46 and Figure 47) only generated about 25,000 transactions. In order to generate more transactions, we decided to rerun the same experiment with 1 MB file transfers in each transaction, which ultimately generated over 300,000 transactions. The first big difference between Figure 48 and Figure 46 is the fact that the machine network connectivity does not seem to predominate as much the GridFTP client performance. We notice many smaller bubbles although some of them have relatively low latencies and relatively high network bandwidth. We also see that Figure 49 depicts a standard deviation that is on average larger than before and much more consistent across the board from well connected to poorly connected nodes. The difference in fairness can be attributed to the fact that having smaller file uploads per transaction significantly increased the server's utilization. Each transaction starts up a GridFTP client, creates a TCP connection to the GridFTP server, transfers a file, and tears down the connection. The difference between having 10 MB files and 1 MB file transfers changed the number of new connections at the server from 2.5 connections per second to 25 connections per second. This would normally not be an issue, but with the server being pushed to its limits with 1300 concurrent clients transferring an aggregate 40 MB/s of data, the increase in connections per second was enough to decrease the fairness of the server.

Overall, the GridFTP server seems to allocate resources fairly across large number of clients (1000s), and across a heterogeneous WAN environment.

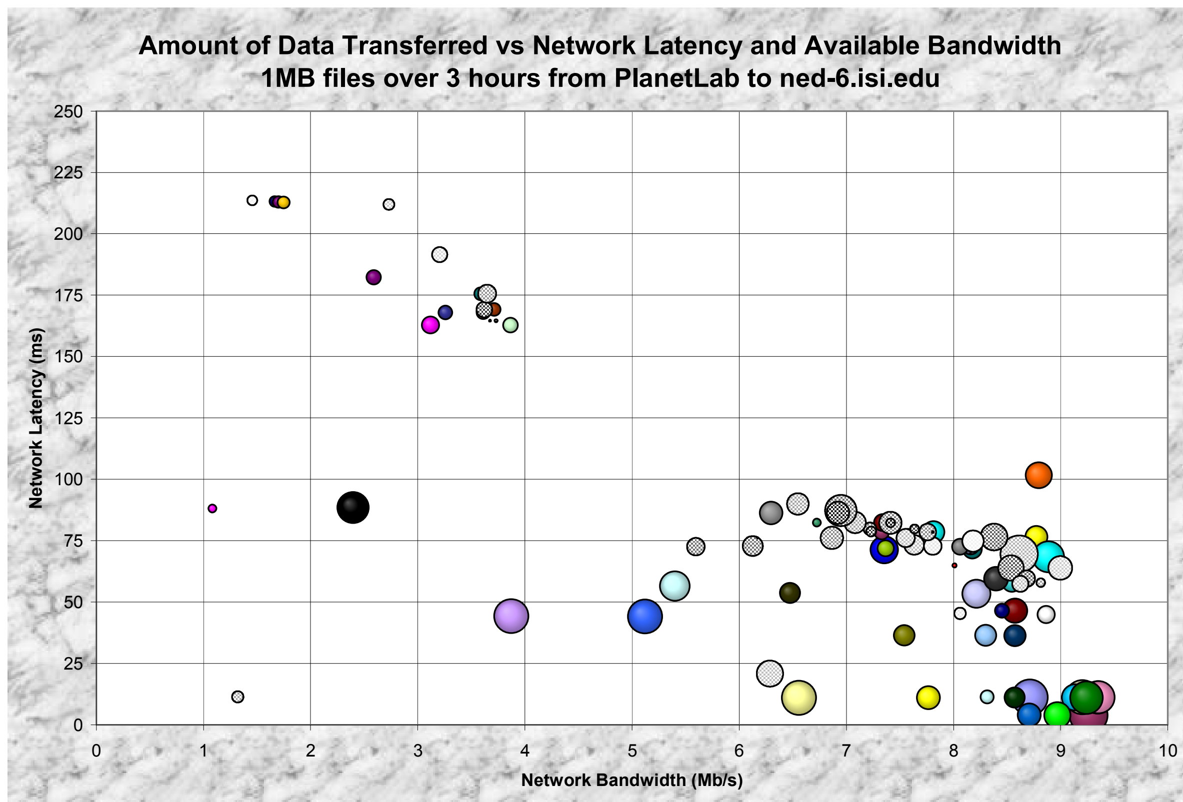


Figure 48: GridFTP resource usage per machine using 1MB files

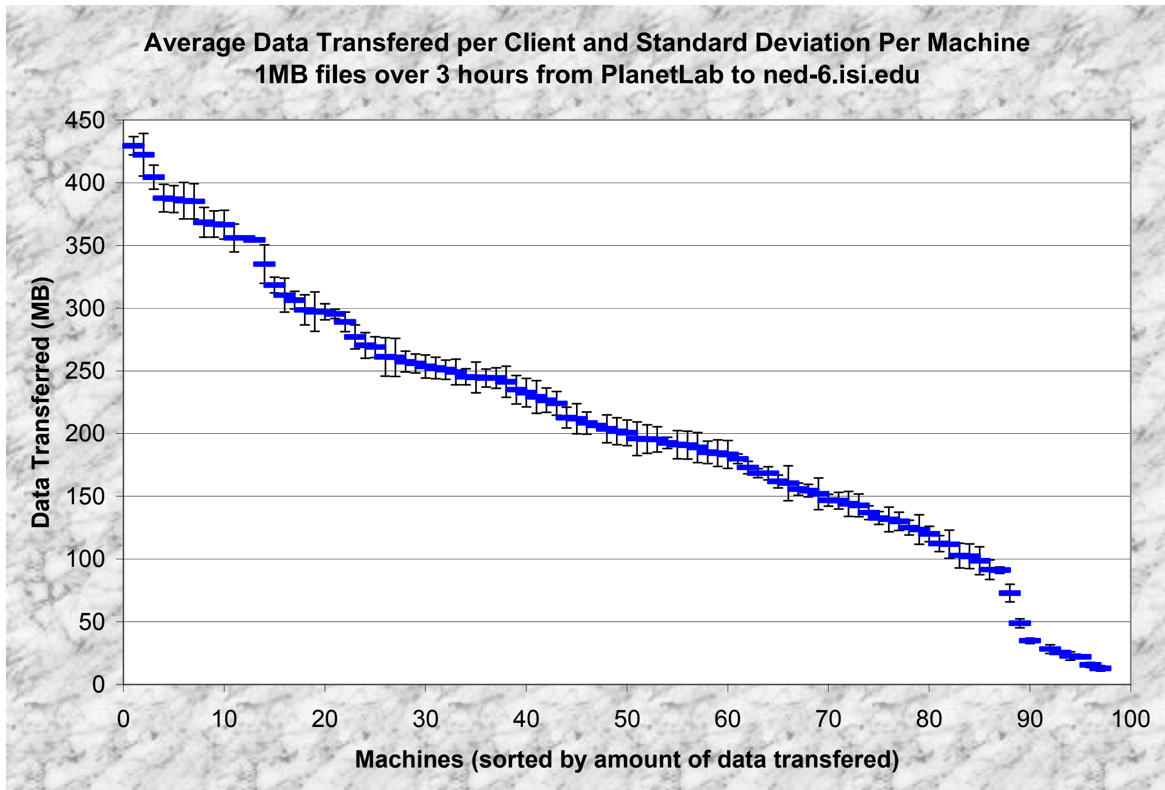


Figure 49: GridFTP resource usage per client using 1MB files

5.2.3 GridFTP Conclusions

We used DiPerf to deploy Zebra clients on multiple hosts and to collect performance data. The GridFTP server (located in Los Angeles, CA) was a 2-processor 1125 MHz x86 machine running Linux 2.6.8.1 with Web100 patches, 1.5 GB memory and 2 GB swap space, 1 Gbit/s Ethernet network connection and 1500 B network MTU. The clients were created on hosts distributed over PlanetLab and at the University of Chicago (UofC).

We were able to run experiments with up to 1800 clients mapped in a round robin fashion on 100 PlanetLab hosts and 30 UofC hosts. The experiments lasted on the order of hours, and normally involved repeated requests to transfer a 10 Mbyte file from the server's disk to the client's /dev/null. Each experiment typically transferred 100 to 1000 Gbytes of data between all the clients to the server. Table 5 summarizes the performance of the GridFTP server in the various test case studies we performed using 100 to 1800 GridFTP clients distributed all over the world.

The results we obtained are encouraging; the server was able to sustain 1800 concurrent clients with just 70% CPU and 0.94 Mbyte memory per request. Furthermore, CPU usage, throughput, and response time remain reasonable even when allocated memory exceeds physical memory, meaning that paging is occurring. Total throughput reaches 25 Mbyte/s with less than 100 clients and exceeds 40 Mbyte/s with around 600 clients. Note that if we actually had 1800 client running on 1800 separate machines (in contrast with 1800 clients running on 130 machines), it is likely that we would have saturated the server much sooner than 600 clients, and perhaps we might have even achieved a higher aggregate throughput. Keep in mind that the zebra clients are heavy weight in the sense that 1 client on 1 machine can most likely use up all available network bandwidth, and hence any increase in clients on the same physical machines will not results in an increase of aggregate throughput. Furthermore, we were able to show that the GridFTP server offered fair resource allocation to clients proportional to each client's available bandwidth; we were also able to show that multiple clients running on the same machine received relatively equal share of resources.

Table 5: GridFTP WAN performance summary ranging from 100 clients up to 1800 clients

Experiment Description	Throughput (MB/s)					Load at Service Saturation	Response Time (sec)				
	Min	Med	Aver	Max	Std. Dev.		Min	Med	Aver	Max	Std. Dev.
Figure 40: 100 clients - 100 hosts	27	30.8	30.7	34.6	1.28	100*	9.3	16.5	28.3	1179	44.4
Figure 41: 500 clients - 100 hosts	33.4	39	39	44.1	1.64	110	25	67	117.5	1833	117.6
Figure 42: 1100 clients - 130 hosts	31.8	37.4	37.3	42.5	1.87	900	52	195	249.4	1594	177.2
Figure 44: 1300 clients - 100 hosts	23.4	33.3	33.2	40.5	2.33	106	69.2	289.4	338.2	3403	252
Figure 45: 1800 clients - 130 hosts	32	38.7	38.6	46.1	2.58	900	52.5	478.3	407.5	1930	247.5

5.3 WS-MDS Index Scalability and Performance Results

We evaluated the WS-MDS index bundled with the Globus Toolkit 3.9.5 on a machine at UChicago. The metrics collected (client view) by DiPerF are:

- **Service response time** or time to serve a query request, that is, the time from when a client issues a request to when the request is completed
- **Service throughput**: aggregate number of queries per second from the client view
- **load**: number of concurrent service requests

We ran our experiments on four different configurations: 1) LAN tests with 4 machines connected via 1 Gb/s from the DiPerF cluster; 2) WAN tests with 128 machines from PlanetLab connected via 10 Mb/s; 3) WAN tests with 288 machines from PlanetLab connected via 10 Mb/s; and 4) LAN+WAN tests with 3 machines in a LAN and 97 machines in a WAN. We ran the WS-MDS index at UChicago on a machine named "m5" with the specs outlined in **Error! Reference source not found.**; the DiPerF framework ran on an AMD K7 2.16GHz with 1GB RAM, 1 Gb/s network connection locally, and 100 Mb/s network connection externally located at UChicago. For each set of tests, the caption below the figure will address the particular configuration of the DiPerF controller and the testbed configuration which yielded the respective results.

5.3.1 WS-MDS Results

In the figures to follow (Figure 50 – Figure 57), each series of points representing a particular metric is also approximated using a moving average over a 60 point interval; since most raw samples were taken once a second, the moving average represents the average over 1 minute intervals. Each figure also has a summary of the results in a table in the upper right hand corner of each figure; the table contains the experiment length in time, number of queries processed by the WS-MDS Index, and the minimum/median/average/maximum of both collected metrics, namely throughput (queries per second), and response time (ms).

Figure 50 represents the performance of the WS-MDS index with no security when 4 clients in a LAN environment targeted the WS-MDS index. Note the large spikes in response times at the beginning of the experiment; each spike corresponds to a new client starting, and they generally occur only for the first query; the summary table gives us a rough idea of how long these peaks were, with the highest being over 2 seconds. The throughput in terms of transactions per second reached as high as 461 queries per second, although for the majority of the experiment it was mostly between 300 and 350 queries per second. Each of the four clients could generate between 80 to 130 queries per second, so the aggregate performance along with the fact that the processors on the WS-MDS Index were only utilized about 60%, it leads us to believe that the WS-MDS index could achieve even higher throughput. The network throughput we observed seemed to be between 8Mb/s and 12Mb/s at the WS-MDS Index, so we do not believe that the network capacity of 1Gb/s was an issue. The same exact experiment produced very similar results even when the LAN network connectivity was downgraded to 100Mb/s.

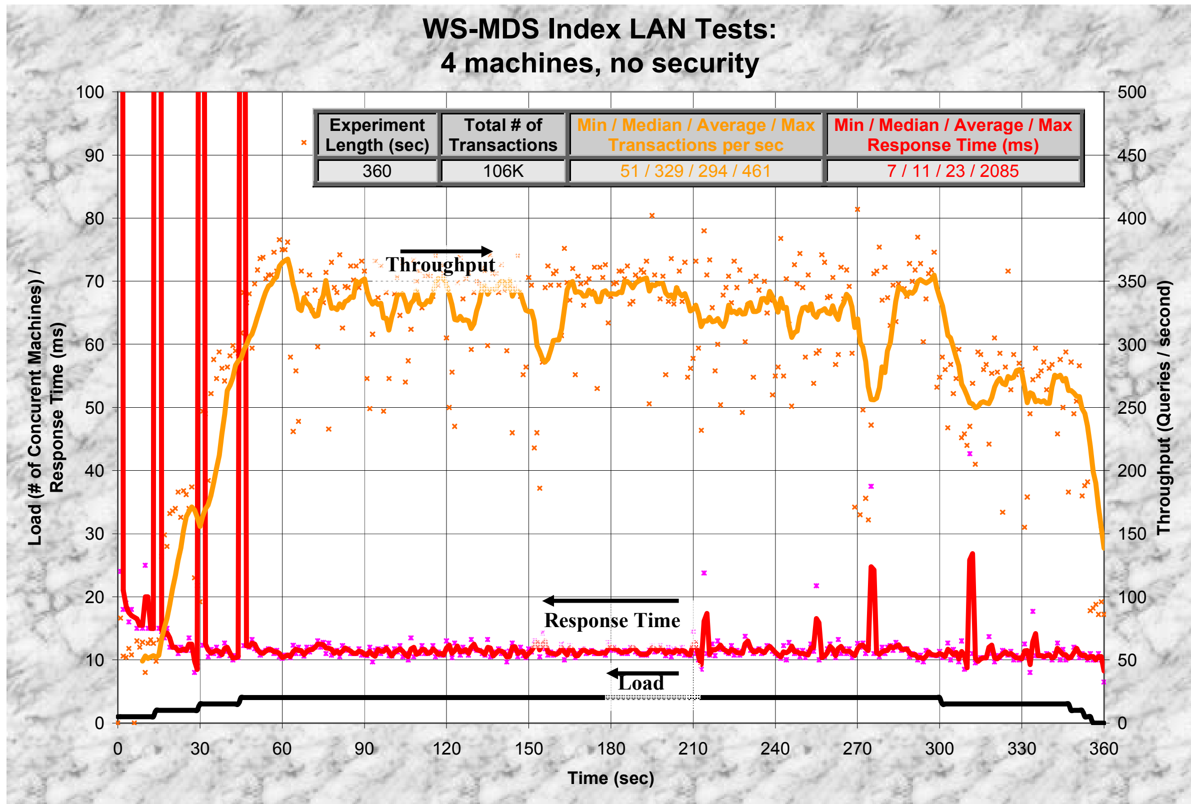


Figure 50: WS-MDS Index LAN Tests with no security including 4 clients running on 4 physical nodes at UChicago in a LAN connected via 1 Gb/s links; tunable parameters: utilized 4 concurrent clients, with each client starting every 15 seconds; left axis – load, response time; right axis – throughput

Figure 51 represents a very similar test to Figure 50, except that we ran 100 clients over the same 4 machines in a LAN. Note the same behavior in response times at the beginning of the experiment, in which there are very large spikes in response times as new clients join.

The new clients joining (at a rate of 3 per second) seem to significantly affect the ability of the WS-MDS Index to process queries; the throughput starts out strong, but it starts to drop almost reaching 0 in the beginning period in which many clients are joining in the experiment. During the period of low throughput due to new clients joining, the CPU utilization was very low, with instances when it would reach an idle state for entire seconds at a time. Otherwise, the only other difference between this experiment and the one in Figure 50 is that the throughput reached almost 500 queries per second with an average of just below 400 transactions per second. The CPU utilization was closer to saturation with 80~90% utilization. The network usage was around 12Mb/s and 15Mb/s.

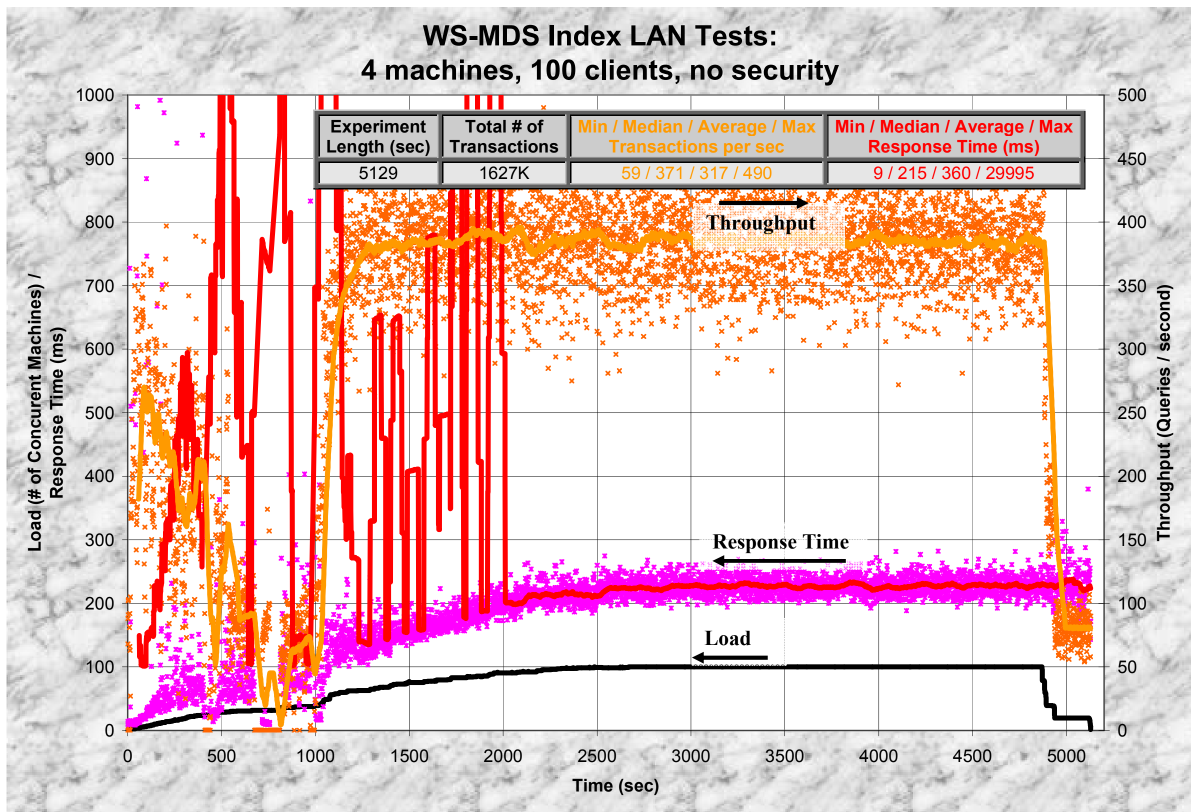


Figure 51: WS-MDS Index LAN Tests with no security including 100 clients running on 4 physical nodes at UChicago in a LAN connected via 1 Gb/s links; tunable parameters: utilized 100 concurrent clients, with each client starting every 15 seconds; left axis – load, response time; right axis – throughput

Figure 52 represents a larger experiment including 128 nodes from PlanetLab in a WAN environment that has network bandwidth connectivity of 10Mb/s and network latencies of 60 ms on average, and as high as 200 ms on some nodes. The maximum throughput achieved was just under 500 queries per second, with close to 400 queries per second on average during the peak portion of the experiment when all 128 clients were accessing the WS-MDS Index simultaneously. It is interesting to note that the response time only increased from the low 200 ms to the high 200 ms. The response time during the peak portion of the experiment when there were 128 machines actively querying the WS-MDS Index seems very similar (248 ms vs. 215 ms) to the performance of the same experiment performed in a LAN as presented in Figure 51; the 15% longer response times in the WAN tests could be attributed simply to the fact that there were 30% more clients actively participating. We observed that the CPU utilization and network bandwidth usage were similar to the LAN tests from Figure 51.

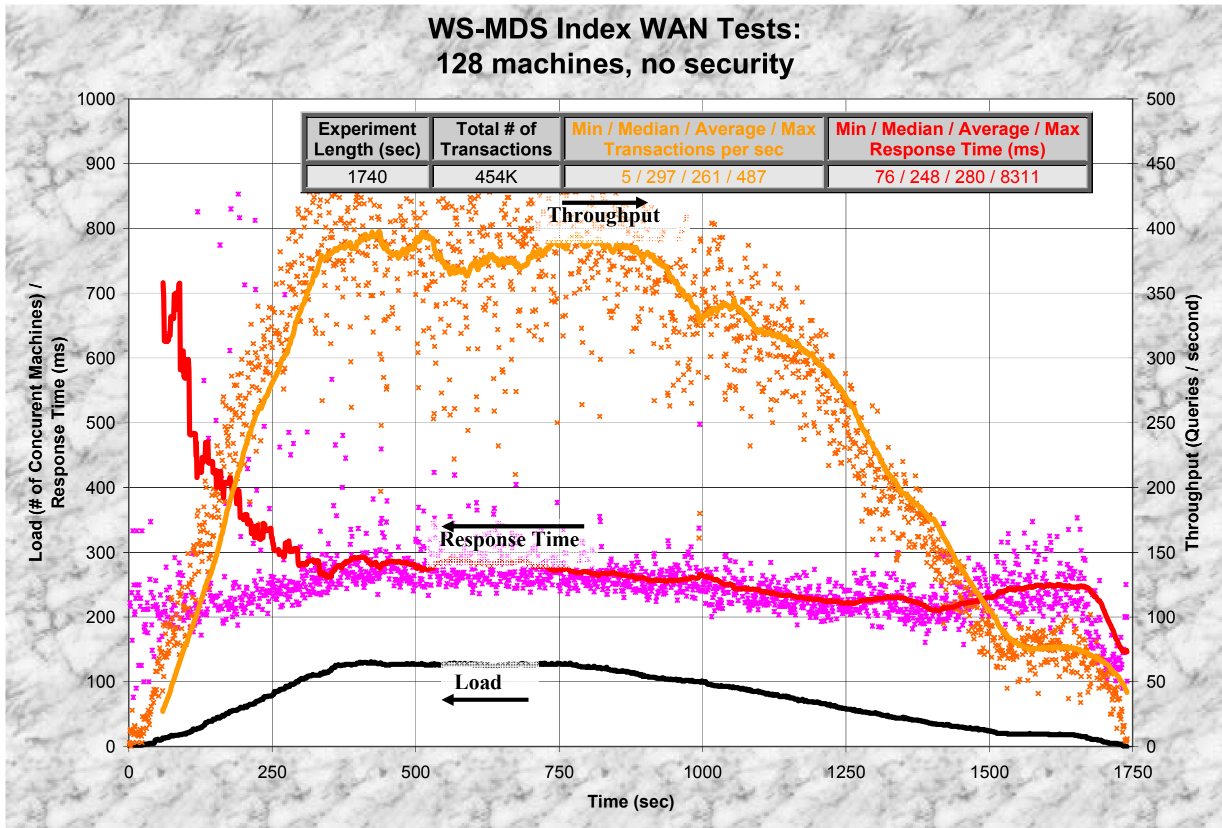


Figure 52: WS-MDS Index WAN Tests with no security including 128 clients running on 128 physical nodes in PlanetLab in a WAN connected via 10 Mb/s links; tunable parameters: utilized 128 concurrent clients, with each client starting every 3 seconds; left axis – load, response time; right axis – throughput

Figure 53 represents another experiment utilizing PlanetLab, but this time we used 288 machines all over the world instead of the 128 machines from the USA.

This test was very interesting due to the fact that the throughput achieved while all 288 machines were concurrently accessing the index was around 200 queries per second on average (when compared to almost 400 queries per second that we achieved from only 128 machines). As the number of machines started dropping, the throughput started to increase, and by about 200 clients, the throughput reached a level more similar to what we had seen in previous tests. Although the WS-MDS Index managed to service all the 288 clients concurrently, its efficiency in terms of sustaining a high throughput clearly dropped.

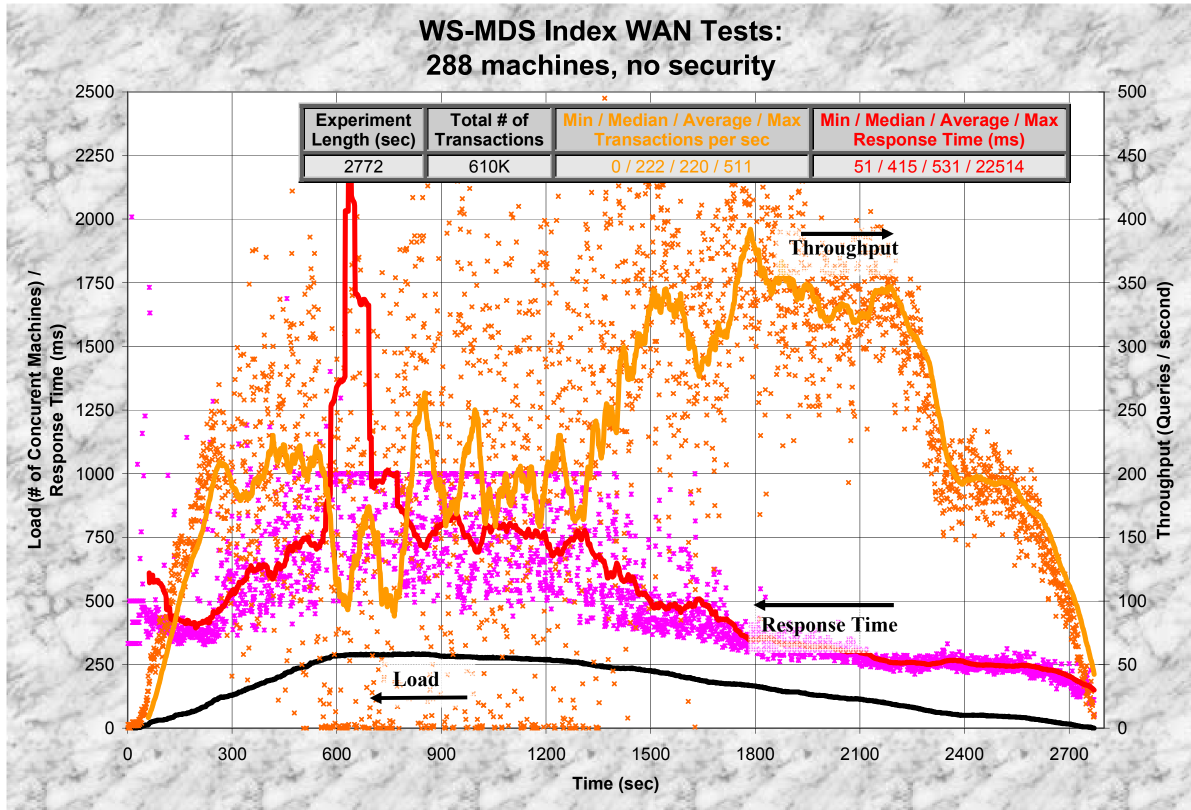


Figure 53: WS-MDS Index WAN Tests with no security including 288 clients running on 288 physical nodes in PlanetLab in a WAN connected via 10 Mb/s links; tunable parameters: utilized 288 concurrent clients, with each client starting every 2 seconds; left axis – load, response time; right axis – throughput

Figure 54 and Figure 55 represent the same experiment in which we used 97 machines from PlanetLab in a WAN setting, and 3 machines at UChicago in a LAN setting. Figure 50 and Figure 52 both showed that each LAN and WAN individually could achieve 300 to 400 queries per second independently. Figure 53 did not show any improvement in the achieved throughput (it actually decreased), from which we concluded that the index could efficiently handle so many clients in a WAN environment. Figure 54 and Figure 55 tries to capture the peak performance of the WS-MDS Index with a testbed that we know could generate more queries per second than we actually observed in this experiment. Ultimately, we obtained performance similar to that of Figure 51 and Figure 52 in which we had 100 clients in a LAN and 128 clients in a WAN respectively. One of our conclusions after these series of tests is that the WS-MDS Index peak throughput on the specific hardware and OS we ran it on is 500 queries per second. Our second conclusion is that a WAN environment can achieve comparable numbers to that of LAN environments given a large enough pool of machines. On the other hand, at least in a WAN environment, it seems that the efficiency of the index decreases after a critical mass of clients is reached; this critical number of clients seems to be in the area of 100 to 200 clients for our particular hardware that we ran the WS-MDS index on and the specific characteristics of the testbed utilized.

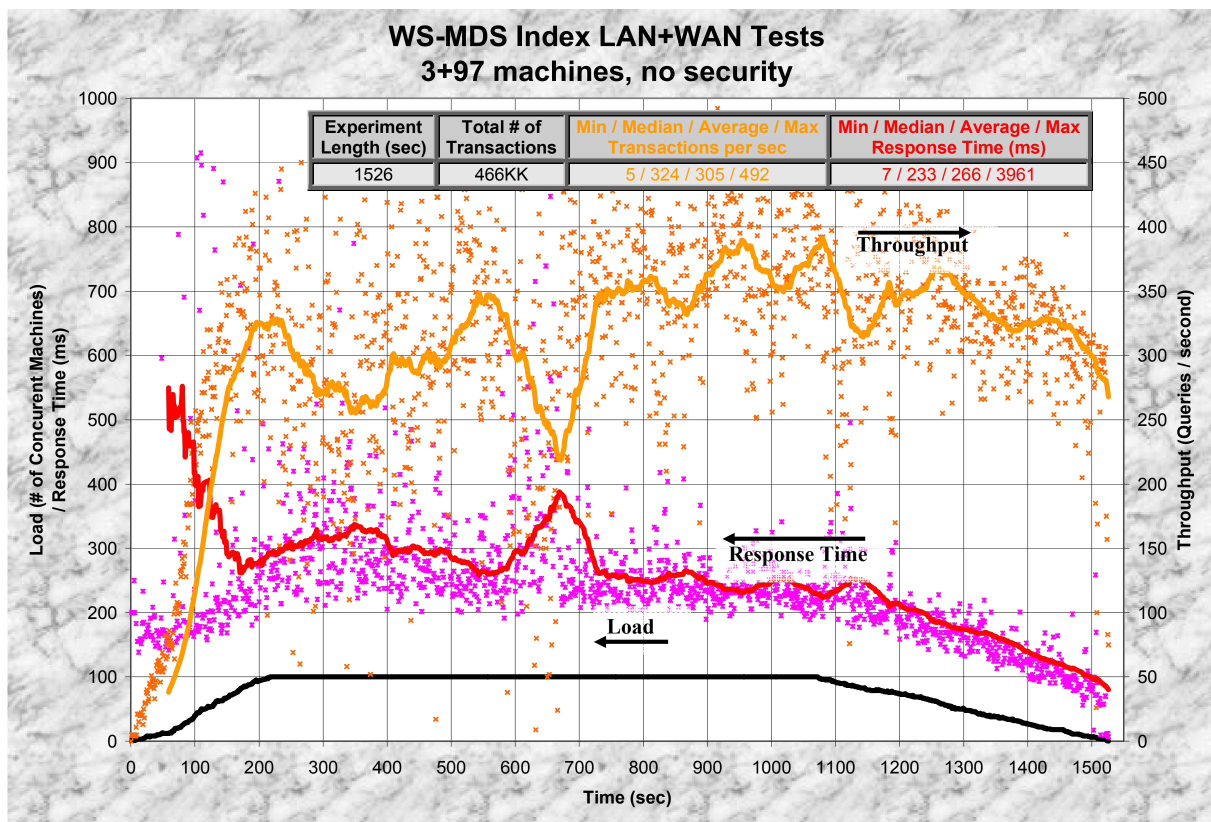


Figure 54: WS-MDS Index LAN+WAN Tests with no security including 100 clients running on 3 physical nodes at UChicago (LAN) and 97 physical nodes in PlanetLab (WAN); tunable parameters: utilized 100 concurrent clients, with each client starting every 2 seconds; left axis – load, response time; right axis – throughput

The next interesting thing we wanted to extract from this experiment was how much did the WAN clients contribute towards the overall achieved throughput and how much we could attribute to the LAN clients. Just as a reminder, the 97 PlanetLab WAN clients were connected via 10Mb/s links with network latencies between 20 ms and 200 ms and an average of 60 ms. On the other hand, the 3 LAN clients were connected via 1Gb/s links and 0.1 ms network latencies. Figure 55 represents our findings about how much the LAN clients and the WAN clients each contributed towards the overall throughput.

Figure 55 shows that for the peak portion of the experiment when all 100 clients were running in parallel, the LAN clients (3% of all clients) generated 5% of the queries. During this period, the 3 LAN clients generated on average 18 queries per second, while in Figure 50 we observed that 4 LAN clients could achieve a throughput well over 350 queries per second. Note how the LAN throughput increases as the WAN clients stop participating, eventually reaching close to 500 queries per second.

This behavior is very peculiar, and it shows the effects of a particular design choice of the developers. Based on the results of these experiments, we concluded that under heavy load, the WS-MDS Index acts as round robin queue. This caused all clients (irrespective of the fact that some were very well connected while others were not) to get equal share of resources. We also observed that the achievable throughput is lower when there are new clients joining. The initial connection is expensive on the service side that it affects the index's capacity to process queries; there seem to be some concurrency issues especially since the processors do not seem heavily loaded in the beginning part of the experiments when many clients were joining the experiment, although the achieved throughput was relatively low.

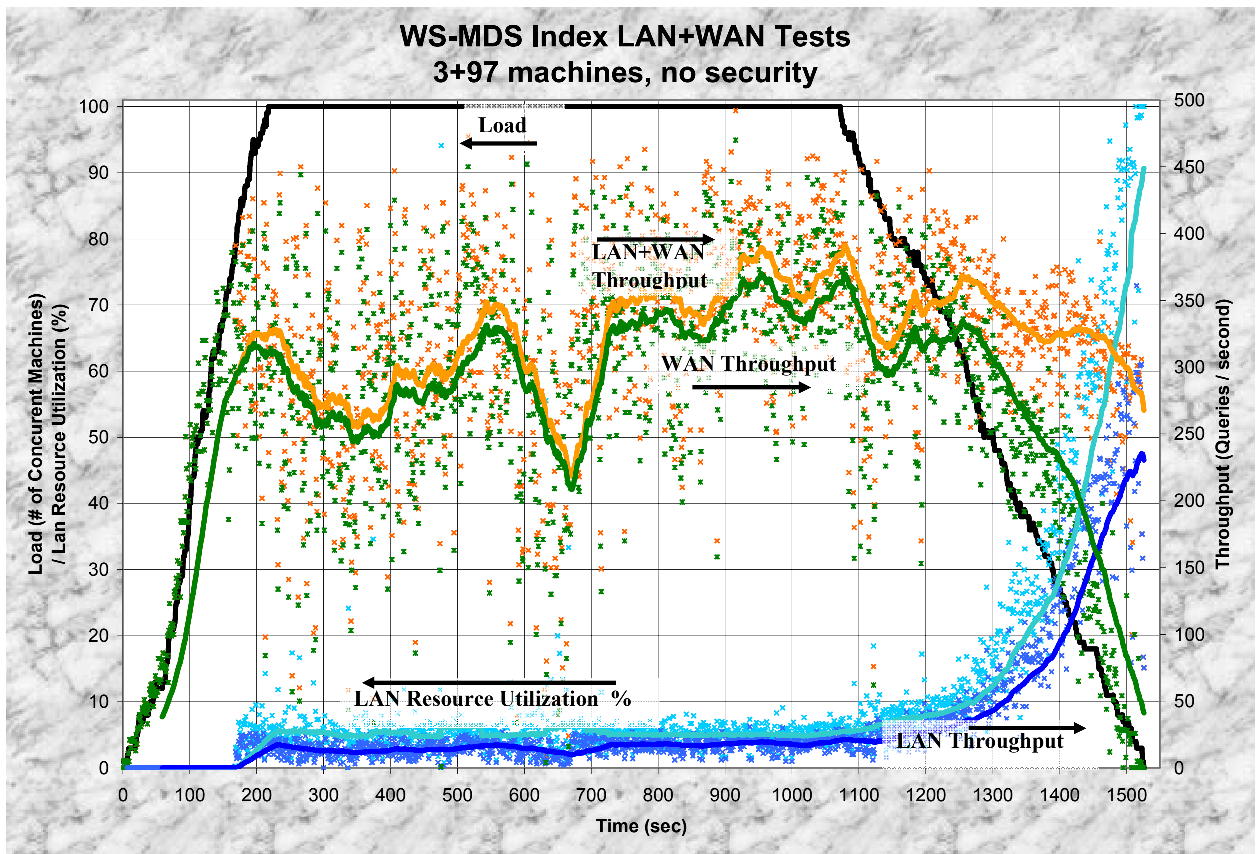


Figure 55: WS-MDS Index LAN+WAN Tests with no security including 100 clients running on 3 physical nodes at UChicago (LAN) and 97 physical nodes in PlanetLab (WAN); tunable parameters: utilized 100 concurrent clients, with each client starting every 2 seconds; left axis – load, LAN resource utilization %; right axis – throughput

Both Figure 56 and Figure 57 cover experiments against the WS-MDS Index with security enabled. Figure 56 was a test performed in a LAN environment which achieved a throughput of 45 queries per second generated by 4 clients at UChicago. Note the response times of about 80 ms per query. Unlike the results from the WAN vs. LAN tests without security in which we obtained similar throughput capacity of the index, the WAN tests with 128 clients proved to have significantly lower performance. The WAN test resulted in a throughput of about 20 queries per second, and with response times on the order of 5 to 7 seconds. Overall, adding security to the WS-MDS Index queries seems to have a significant performance penalty.

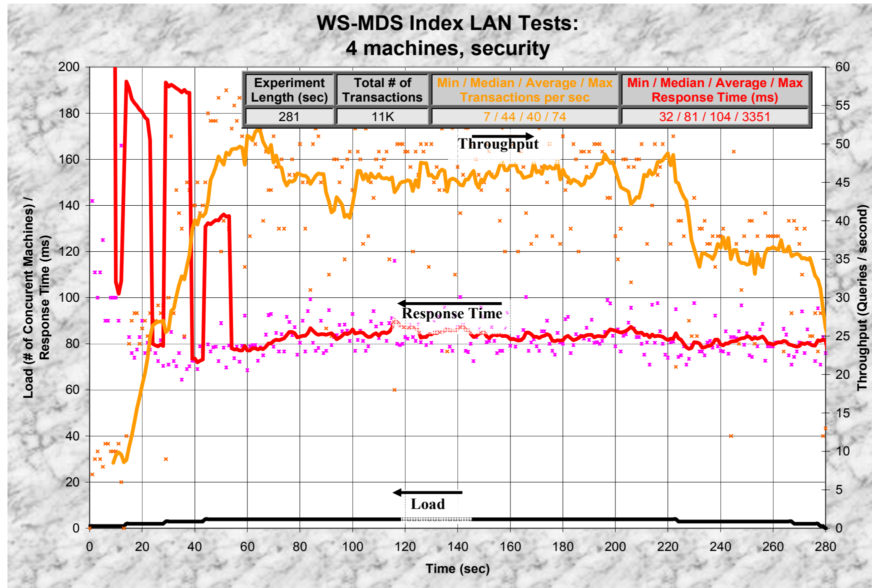


Figure 56: WS-MDS Index LAN Tests with security including 4 clients running on 4 physical nodes at UChicago in a LAN connected via 1 Gb/s links; tunable parameters: utilized 4 concurrent clients, with each client starting every 15 seconds; left axis – load, response time; right axis – throughput

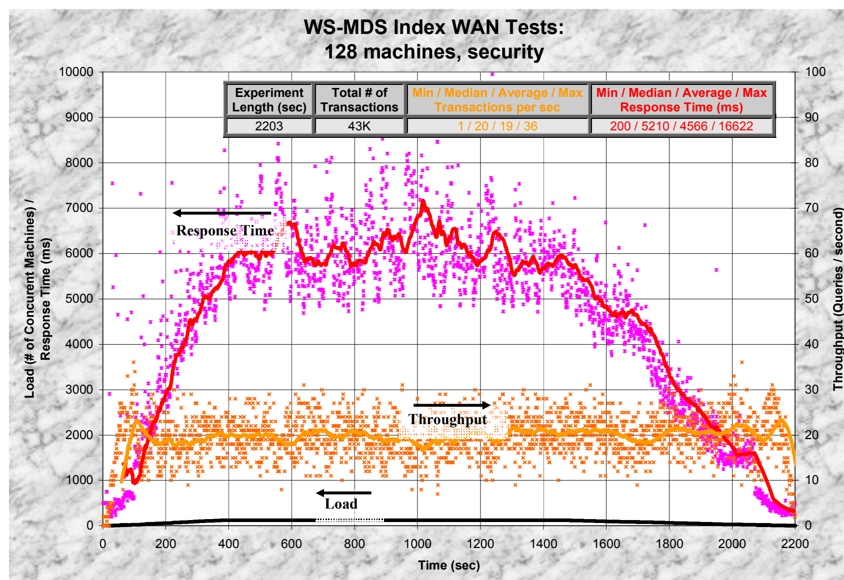


Figure 57: WS-MDS Index WAN Tests with security including 128 clients running on 128 physical nodes in PlanetLab in a WAN connected via 10 Mb/s links; tunable parameters: utilized 128 concurrent clients, with each client starting every 3 seconds; left axis – load, response time; right axis – throughput

5.3.2 WS-MDS Conclusions

Table 6 shows the summary of the main results from the WS-MDS experiments performed in LAN and WAN with both security enabled and disabled.

Table 6: WS-MDS summary of experiments in both LAN and WAN with both security enabled and disabled; for the "load at service saturation, the * indicates that the service was not saturated with the number of concurrent clients that were used

Experiment Description	Throughput (trans/sec)				Load at Service Saturation	Response Time (ms)			
	Min	Med	Aver	Max		Min	Med	Aver	Max
Figure 50: LAN 4	51	329	294	461	4*	7	11	23	2085
Figure 51: LAN 4-100	59	371	317	490	60	9	215	360	29995
Figure 52: WAN 128	5	297	261	487	128*	76	248	280	8311
Figure 53: WAN 288	0	222	220	511	225	51	415	531	22514
Figure 54: WAN 97 + LAN 3	5	324	305	492	100*	7	233	266	3961
Figure 56: LAN 4 + security	7	44	40	74	4*	32	81	104	3351
Figure 57: WAN 128 + security	1	20	19	36	22	200	5210	4566	16622

With no security, WAN throughput performance was similar to that of LAN throughput performance given a large enough set of clients. On the other hand, with security enabled, the WAN tests showed less than half the throughput when compared to a similar test from a LAN. The initial query for a connection could take a very long time (we observed times as high as 30 seconds, and was almost always greater than a second); furthermore, many new connections adversely affects the efficiency of the index and its ability to process as many queries as it could have if it weren't for the new connections. During the start-up phase of the experiments when many clients would be making new connections, the throughput would be relatively poor, and the CPU utilization would be low as well. Another observation is that under heavy load, the WS-MDS index acts as a round robin queue which effectively distributes the share of resources evenly across all clients irrespective of the connectivity of the client.

5.4 Grid Services

5.4.1 GT3.2 Instance Creation

We have used DiPerF to perform tests on service instance creation in a Globus Toolkit 3 service (similar to the DI-GRUBER implementation), and found a peak throughput of about 14 requests per second. Average service response time under 'normal' load was about 4s. Average service response time under 'heavy' load was about 10 seconds. We also observed that under heavy load the WS service does not allocate resources evenly among clients [13]. The results of Figure 58 show how the service response time increases with the number of concurrent machines. As a consequence, there is a real need to investigate other ways of building and organizing a scheduling infrastructure for large grids with many submitting hosts, and to understand the implications this has for performance, reliability and scheduling decision accuracy.

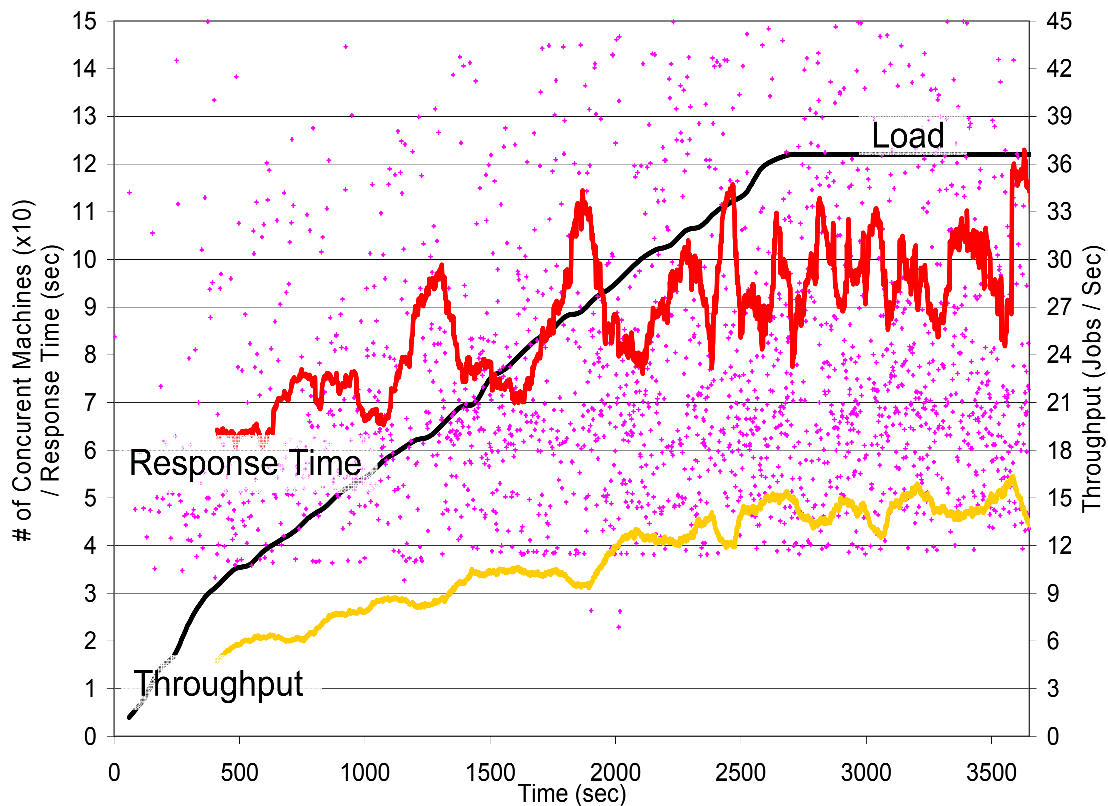


Figure 58: GT3.2 Service Instance Creation: Response time, Throughput, and Load

For example, DI-GRUBER bypasses the OGSA service instance creation latency by providing the possibility of using generic and long term clients. One client is enough to handle all jobs submissions from at least six submission hosts. Unfortunately, in a large environment with many submission hosts, the performance problem still remains due to the high number of GRUBER clients that can run in parallel.

5.4.2 DI-GRUBER: a distributed grid service

Managing usage Service Level Agreements (SLA) within large environments that integrate participants and resources spanning multiple physical institutions is a challenging problem. Maintaining a single unified usage SLA management decision point over hundreds to thousands of jobs and sites can quickly become a problem in terms of reliability as well as performance. Dumitrescu et al. [24, 89, 90] developed GRUBER, a grid Usage SLA-based resource broker; an extension to GRUBER, called DI-GRUBER, is a distributed grid Usage SLA-based resource broker that allows multiple decision points to coexist and cooperate in real-time. DI-GRUBER ultimately addresses issues regarding how usage SLAs can be stored, retrieved and disseminated efficiently in a large distributed environment. The key question this performance study addresses is the scalability and performance of DI-GRUBER in large Grid environments. We conclude that as little as three to ten decision points could be enough in an environment with 300 sites and 60 VOs, an environment ten times larger than today's Grid3.

DI-GRUBER is composed of four principal components, the DI-GRUBER engine, the DI-GRUBER site components, the DI-GRUBER site selectors, and the DI-GRUBER queue manager. In the DI-GRUBER prototype, usage SLAs are specified through a specialized interface. The main components that we concentrate our performance measurements are the engine and the site selectors, as they are the main elements in providing adequate scheduling decisions when resources are available.

5.4.2.1 Experimental Settings & Performance Metrics

Performance Metrics: We consider several metrics to evaluate the effectiveness of DI-GRUBER in practice. We evaluate the effectiveness of different decision point deployment infrastructures by measuring *Average Response Time* (Response) and *Average Throughput* (Throughput) as a function of the environment complexity. All metrics are important as a good infrastructure will maximize delivered resources and meet owner intents.

We define Response as follows, with RT_i being the individual job time response:

- Response = $\Sigma_{i=1..N} RT_i / N$
- Throughput is defined as the number of requests completed successfully by the service averaged over a short time interval (per second or minute). We used in our measurements a second as the interval of measure.

DI-GRUBER Infrastructure: We used one to ten DI-GRUBER decision points deployed on machines around the world. Each decision point maintained a view of the configuration of "simulated" global environment, while exchanging information about instantaneous utilizations.

Workloads and Environment: We used composite continuous workloads that overlay work for 60 VOs and 10 VO groups. The submission interval was one hour in all cases. Each submission site scheduled randomly one job to one DI-GRUBER decision point on behalf of a pair (VO, group). The environment was composed of 300 sites (ten times larger than what Grid3 is today). The initial configurations were based on the Grid3 configurations.

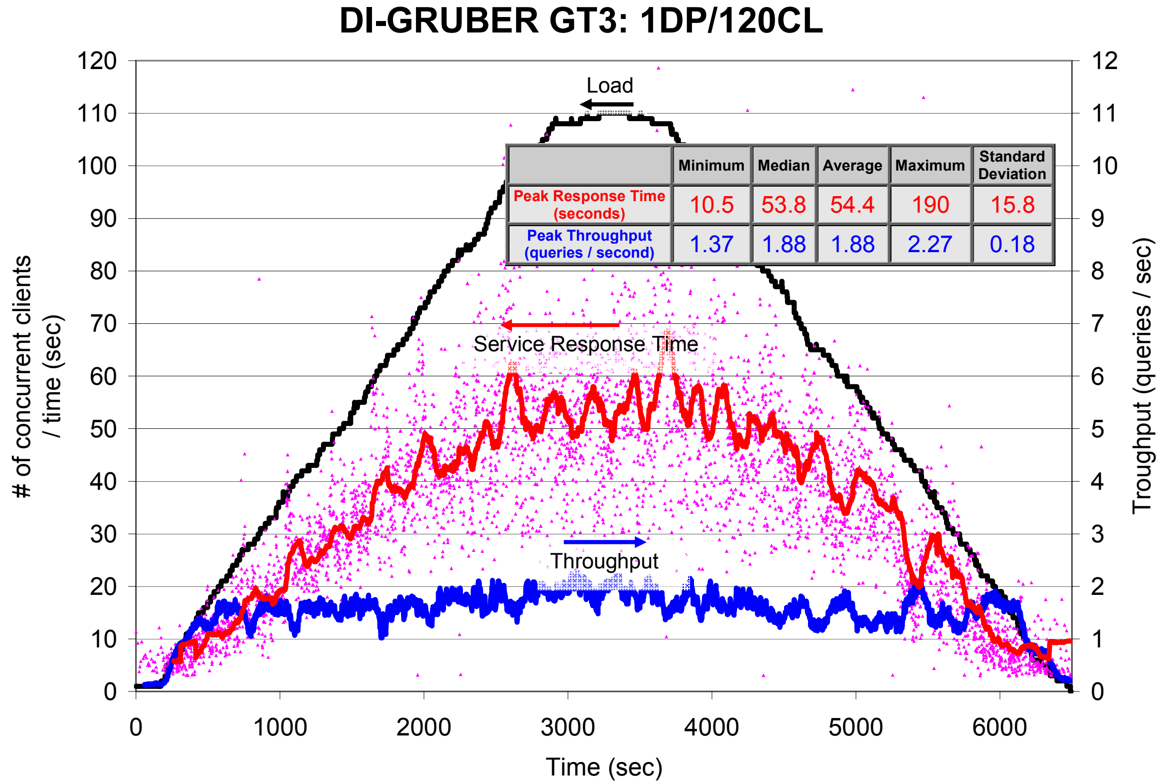
Usage SLA: For each virtual site, we used a similar approach in usage SLA specification by using the Grid3 policies as the initial configurations.

Job states: The workload executions are based on a model where jobs pass through four states: 1) submitted by a user to a submission host; 2) submitted by a submission host to a site, but queued or held; 3) running at a site; and 4) completed.

5.4.2.2 GT3-based Empirical Results

We performed our tests on PlanetLab by deploying DI-GRUBER decision points on several nodes. The complexity of the brokers' network used (the full mesh among the various decision points) in this set of tests is specifically low in order to provide simple case scenario that are tractable in analysis and can easily be followed by readers.

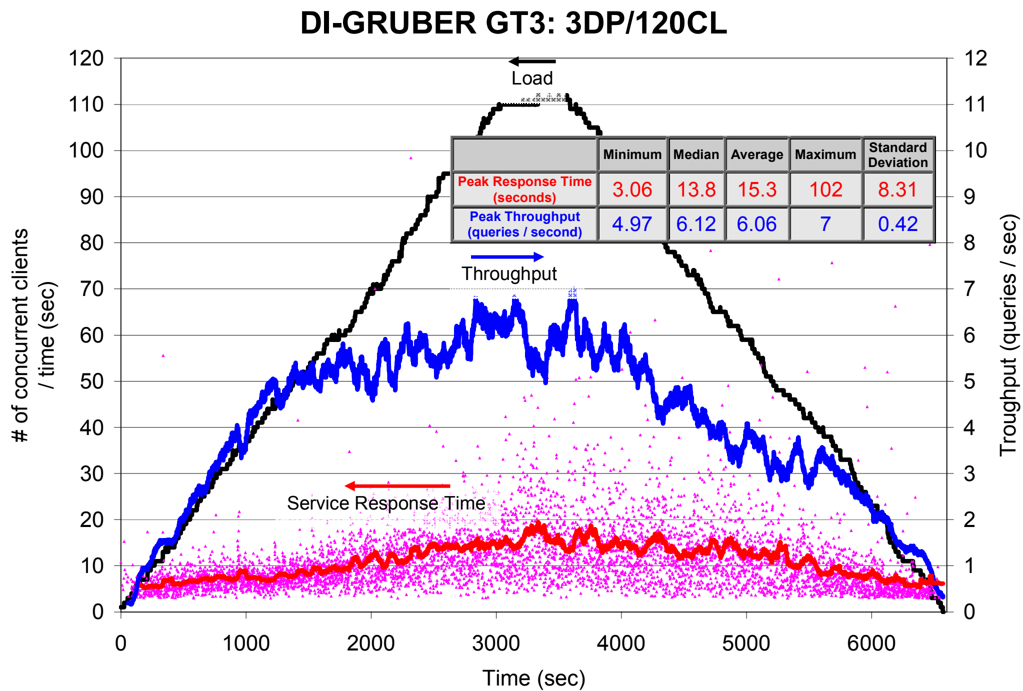
We used DiPerF to slowly vary the participation of 120 clients. Figures below present **Response**, **Throughput** and **Load** as measured by DiPerF. The results of Figure 59 show a steadily increasing service response time as the number of concurrent machines increases; during the peak period, the average service **Response** time was about 54 seconds. **Throughput** increases very rapidly, but after about 15 concurrent clients, it plateaus at a little less than 2 queries per second; the throughput remains relatively constant at about 1.9 queries per second even when all testing machines (120) are accessing the service in parallel.



	Minimum	Median	Average	Maximum	Standard Deviation
Peak Response Time (seconds)	10.5	53.8	54.4	190	15.8
Peak Throughput (queries / second)	1.37	1.88	1.88	2.27	0.18

Figure 59: Centralized GT3-based Scheduling Service

The results in Figure 60 show the performance of DI-GRUBER with 3 decision points as compared to the single decision point tested in **Error! Reference source not found.**. **Throughput** increases slowly and achieves a value of about 6 job scheduling requests per second when all testing machines are accessing the service in parallel. The service **Response** time is also smaller (about 15 seconds) in average compared with the previous results (about 54 seconds); once the number of machines starts decreasing, **Response** does not decrease in a symmetrical way, the service remaining somehow in a state that impedes it to answer as quickly as before. An explanation for this behavior is the fact that once jobs were scheduled for execution, they were considered long term running jobs. Each job was running for a longer time interval than the entire test time and the grid state irreversibly changing (free resources monotonically decreasing).



	Minimum	Median	Average	Maximum	Standard Deviation
Peak Response Time (seconds)	3.06	13.8	15.3	102	8.31
Peak Throughput (queries / second)	4.97	6.12	6.06	7	0.42

Figure 60: GT3-based Distributed Scheduling Service with 3 Decision Points

Our last test involved the performance of DI-GRUBER with 10 decision points; the results are depicted in Figure 61. The average service **Response** time decreased even further to about 10 seconds, and the achieved **Throughput** reached about 8 queries per second during the peak load period.

The distributed service provides a symmetrical behavior with the number of concurrent machines independent of the state of the grid (lightly or heavily loaded). This result verifies the intuition that for a certain grid configuration size, there is an appropriate number of decision points that can serve the scheduling purposes under an appropriate performance constraint.

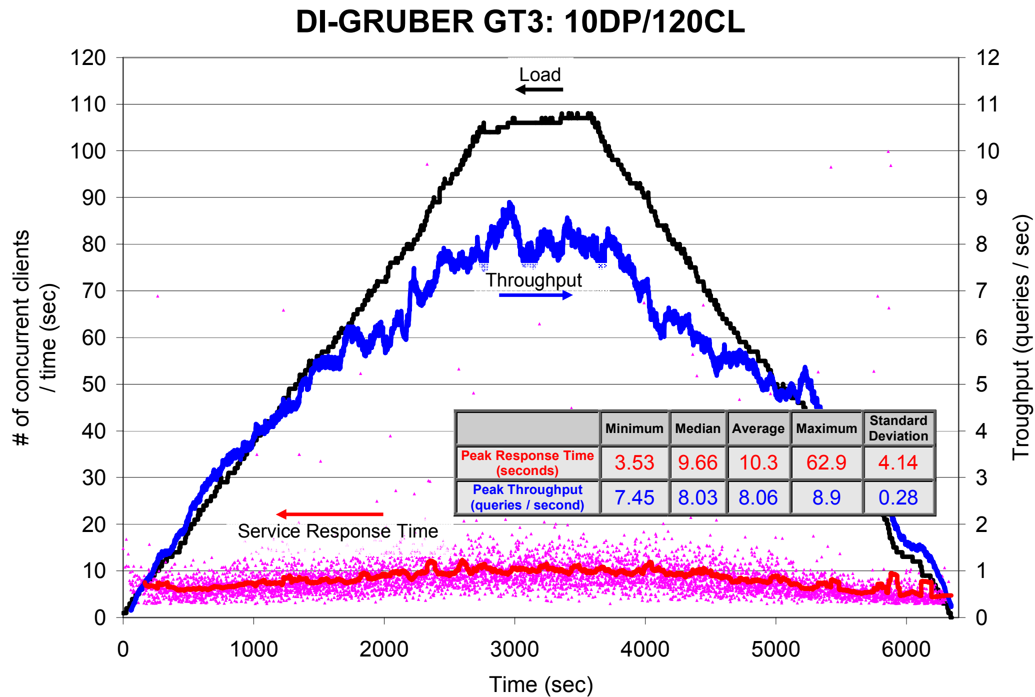


Figure 61: GT3-based Distributed Scheduling Service with 10 Decision Points

The overall improvement in terms of throughput and response time is two to three times when a three-decision point infrastructure is deployed, while for the ten-decision point infrastructure the throughput increased almost five times. Both Table 7 and Table 8 show a summary of the performance of GT3-based DI-GRUBER with respect to response time and throughput. We can clearly see in both tables that the response times dropped significantly and the throughput increased as well as the number of decision points increased. We also see that with 10 decision points, we were not able to saturate the service with more than 100 clients, while with less decision points, we did manage to saturate the service.

Table 7: GT3-based distributed scheduling service summary with 1, 3, and 10 decision points peak response time in seconds

Experiment Description	Minimum	Median	Average	Maximum	Standard Deviation
GT3 1G	10.5	53.8	54.4	190	15.8
GT3 3G	3.06	13.8	15.3	102	8.31
GT3 10G	3.53	9.66	10.3	62.9	4.14

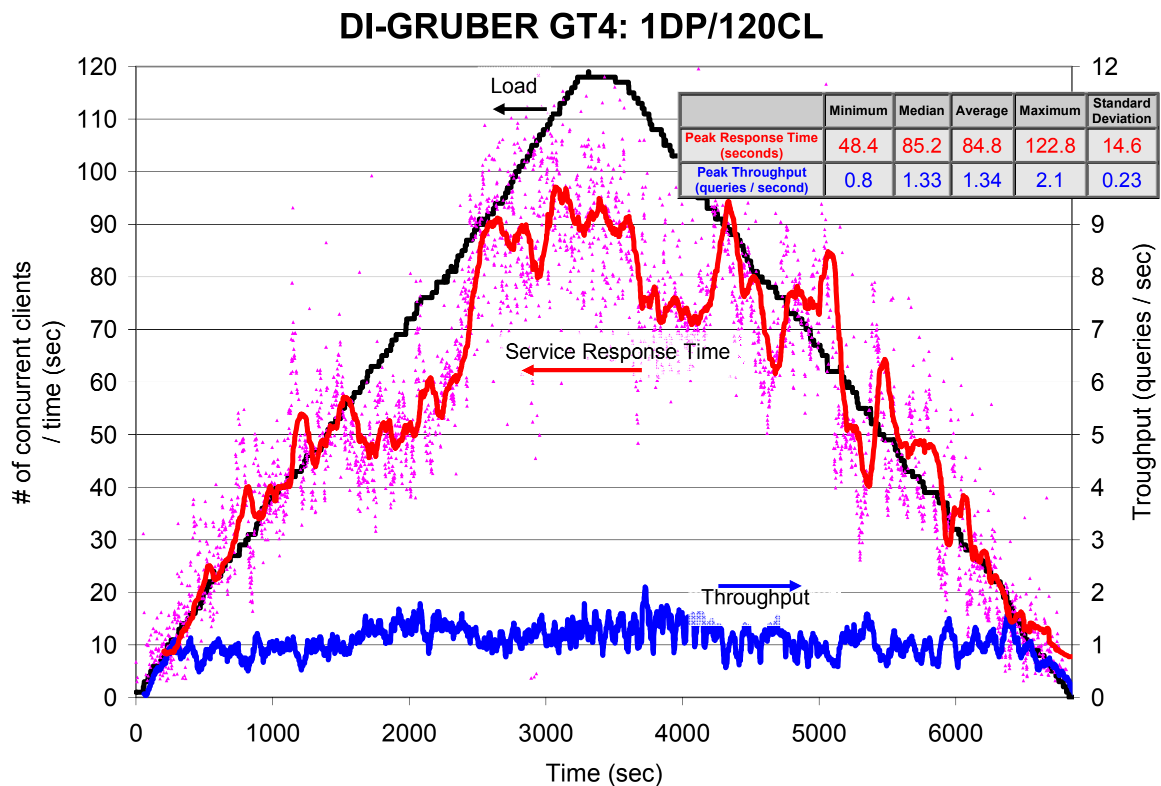
Table 8: GT3-based distributed scheduling service summary with 1, 3, and 10 decision points peak throughput (queries per second) and the load at service saturation (* indicates that the service was not saturated with the number of concurrent clients that were used)

Experiment Description	Load at Service Saturation	Minimum	Median	Average	Maximum	Standard Deviation
GT3 1G	20	1.37	1.88	1.88	2.27	0.18
GT3 3G	50	4.97	6.12	6.06	7	0.42
GT3 10G	105*	7.45	8.03	8.06	8.9	0.28

5.4.2.3 GT4-based Empirical Results

With the release of the Globus Toolkit 4 on the horizon, and the availability of the beta version (GT3.9.5), we ported the DI-GRUBER implementation from GT3 to GT4. These experiments performed with a prerelease of GT4-based implementation of DI-GRUBER are important because the two implementations have different underlying technologies (heavy Grid Services vs. lightweight Web Services), and provide as well a means for further extrapolation of various parameters and behaviors not only based on the environment size, but also with the infrastructure performance. We consider that it is like comparing two different resource brokers built for similar purposes but based on different technologies.

Using DiPerF, we varied slowly the number of clients for this set of tests. Figures below present as before **Response**, **Throughput** and **Load**. The results of Figure 62 show a steadily increasing service response time as the number of concurrent machines increases; during the peak period, the average service **Response** time was about 84 seconds. **Throughput** increases very rapidly, but after about 10 concurrent clients, it plateaus just above 1 query per second; the throughput remains relatively constant at about 1.3 queries per second even when all testing machines (120) are accessing the service in parallel.



	Minimum	Median	Average	Maximum	Standard Deviation
Peak Response Time (seconds)	48.4	85.2	84.8	122.8	14.6
Peak Throughput (queries / second)	0.8	1.33	1.34	2.1	0.23

Figure 62: GT4-based Centralized Scheduling Service

The results in Figure 63 show the performance of the GT4-based DI-GRUBER with 3 decision points as compared to the single decision point tested in Figure 62. **Throughput** increases slowly and achieves a value of about 4 job scheduling requests per second when all testing machines are accessing the service in parallel. The service **Response** time is also smaller (about 26 seconds) in average compared with the previous results (about 84 seconds).

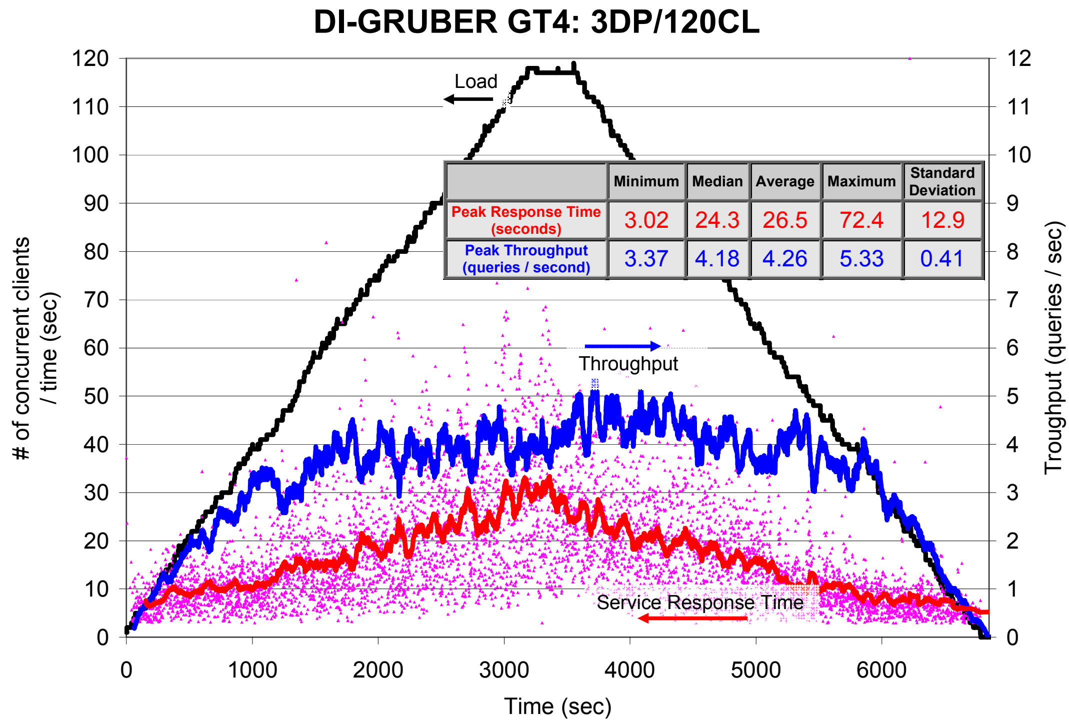


Figure 63: GT4-based Distributed Scheduling Service with 3 Decision Points

Our last test involved the performance of the GT4-based DI-GRUBER with 10 decision points; the results are depicted in Figure 64. The average service **Response** time decreased even further to about 13 seconds, and the achieved **Throughput** reached about 7.5 queries per second during the peak load period. The distributed service provides a symmetrical behavior with the number of concurrent machines independent of the state of the grid (lightly or heavily loaded). This result verifies the intuition that for a certain grid configuration size, there is an appropriate number of decision points that can serve the scheduling purposes under an appropriate performance constraint.

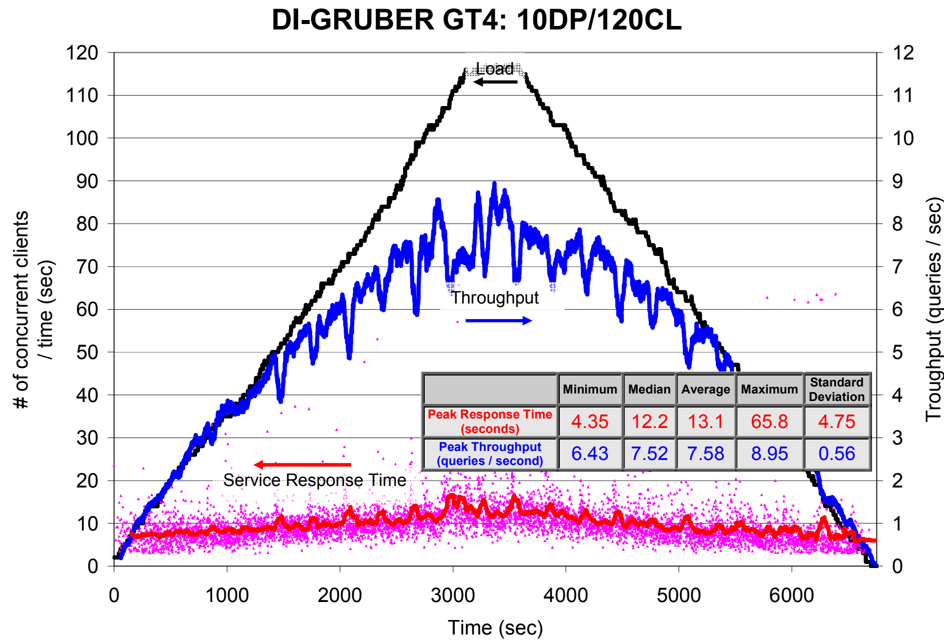


Figure 64: GT4-based Distributed Scheduling Service with 10 Decision Points

The overall improvement in terms of **Throughput** and **Response** is three times larger when a three-decision point infrastructure is deployed, while for the ten-decision point infrastructure performance is five times better. Again, practically, for this DI-GRUBER implementation around or more than three decision points are enough for an optimal response time when around 120 DI-GRUBER clients are scheduling jobs.

Both Table 7 and Table 8 show a summary of the performance of GT4-based DI-GRUBER with respect to response time and throughput. We can clearly see in both tables that the response times dropped significantly and the throughput increased as well as the number of decision points increased. We also see that with 10 decision points, we were not able to saturate the service with more than 100 clients, while with less decision points, we did manage to saturate the service.

Table 9: GT4-based distributed scheduling service summary with 1, 3, and 10 decision points peak response time in seconds

Experiment Description	Minimum	Median	Average	Maximum	Standard Deviation
GT4 1G	48.4	85.2	84.8	122.8	14.6
GT4 3G	3.02	24.3	26.5	72.4	12.9
GT4 10G	4.35	12.2	13.1	65.8	4.75

Table 10: GT4-based distributed scheduling service summary with 1, 3, and 10 decision points peak throughput (queries per second) and the load at service saturation (* indicates that the service was not saturated with the number of concurrent clients that were used)

Experiment Description	Load at Service Saturation	Minimum	Median	Average	Maximum	Standard Deviation
GT4 1G	10	0.8	1.33	1.34	2.1	0.23
GT4 3G	65	3.37	4.18	4.26	5.33	0.41
GT4 10G	115*	6.43	7.52	7.58	8.95	0.56

5.4.2.4 DI-GRUBER Conclusions

Managing usage SLAs within large virtual organizations (VOs) that integrate participants and resources spanning multiple physical institutions is a challenging problem. Maintaining a single unified decision point for usage SLA management is a problem that arises when many users and sites need to be managed. DI-GRUBER offers a solution to address the question on how SLAs can be stored, retrieved and disseminated efficiently in a large distributed environment. *The key question these experiments address is the scalability and performance of the DI-GRUBER scheduling infrastructure in large Grid environments.*

The contribution of this work are the results we achieved on two dimensions – how well our proposed solution (DI-GRUBER) performed in practice (three to ten decision points prove to be enough to handle a grid ten times larger than today's Grid3 [63], with 4 to 5 decision points being sufficient as refined by GRUB-SIM [25]) and a methodology to measure the success of such a meta-scheduler (performance metrics). We also introduced two enhancements to our DI-GRUBER framework, namely the distributed approach in resource scheduling and usage SLA management, and an approach for dynamic computing the number of decision points for various grid settings. This performance study was made possible due to the ease of large scale testing that DiPerF facilitates. More experimental result on DI-GRUBER can be found in [25].

5.5 Other Work that has used DiPerF

The papers or technical reports that used DiPerF along with the place of publishing are:

- DiPerF: an automated DIstributed PERformance testing Framework [82] – Grid2004
- A Scalability and Performance Evaluation of a distributed Usage SLA-based Broker in Large Grid Environments [26] – GriPhyN/iVDGL Technical Report, March 2005
- DI-GRUBER: A Distributed Approach for Grid Resource Brokering [25] – under review at SC 2005
- ZEBRA: The Globus Striped GridFTP Framework and Server [20] – under review
- Performance Measurements in Running Workloads over a Grid [27] – under review at SC 2005
- Extending a distributed usage SLA resource broker with overlay networks to support Large Dynamic Grid Environments [91] – work in progress
- Decreasing End-to-End Job Execution Times by Increasing Resource Utilization using Predictive Scheduling in the Grid [92] – UChicago, Grid Computing Seminar, 2005

Papers or technical reports that have mentioned DiPerF along with the place of publishing are:

- Systems Performance Evaluation Methods for Distributed Systems Using Datastreams [93] – MS Thesis, University of Kansas, January 2005
- Deploying C++ Grid Services: Options and Performance [94] – Duke University, Federated Distributed Systems, December 2004
- Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures [95] – ICSOC 2004

6 Future Work

The Grid has the potential to grow significantly over the course of the next decade and therefore the mechanisms that make the Grid possible need to become more efficient in order for the Grid to scale. One of these mechanisms revolves around resource management; ultimately, there will be so many resources in the Grid, that if they are not managed properly, only a very small fraction of those resources will be utilized. While good resource utilization is very important, it is also a hard problem due to widely distributed dynamic environments normally found in the Grid. It is important to develop an experimental methodology for automatically characterizing grid software in a manner that allows accurate evaluation of the software's behavior and performance before deployment in order to make better informed resource management decisions. Many Grid services and software are designed and characterized today largely based on the designer's intuition and on ad hoc experimentation; having the capability to automatically map complex, multi-dimensional requirements and performance data among resource providers and consumers is a necessary step to ensure consistent good resource utilization in the Grid. This automatic matching between the software characterization and a set of raw or logical resources is a much needed functionality that is currently lacking in today's Grid resource management infrastructure. Ultimately, my proposed work, which addresses performance modeling with the goal to improve resource management, could ensure that the efficiency of the resource utilization in the Grid will remain high as the size of the Grid grows.

Through our previous work, we have shown that DiPerF [82, 26, 20] can be used to model the performance characteristics of a service in a client/server scenario. Using the basic concept of DiPerF, we believe we can also create performance models that can be used to predict the future performance of distributed applications. Modeling distributed applications (i.e. parallel computational programs) might be more challenging since the dataset on which the applications work against often influence the performance of the application, and therefore general predictive models might not be sufficient.

Using DiPerF and a small dedicated cluster of machines, we can build dynamic performance models to automatically map raw hardware resources to the performance of a particular distributed application and its representative workload; in essence, these dynamic performance models can be thought of as job profiles, and will be implemented in the component DiProfile. The intuition behind DiProfile is that based on some small sample workload (with varying sizes) and a small set of resources (with varying size), we can make predictions regarding the execution time and resource utilization of the entire job running over the complete dataset. The DiProfile stage will be a relatively expensive component in both time and computational resources, however its overhead will be warranted as long as the typical job submitted is significantly larger than the amount of time DiProfile needs to build its dynamic performance models.

There is a gap between software requirements (high level) and hardware resources (low level). Automatic mapping could produce better scheduling decisions and give users feedback with the expected running time of their software. Using DiProfile, we can make predictions on the performance of the jobs based on the amount of raw resources dedicated to the jobs. The accuracy of the predictions will heavily rely on the idea that reliable software performance characterization is possible with only a fraction of the data input space.

Using DiPred and user feedback (or even user specified high level performance goals), the scheduler (DiSched) can make better decisions to satisfy the requested duration of the job, where the job should be placed, etc. Since jobs are profiled based on what raw resources they will likely consume and the duration of those resource usage, multiple different jobs could be simultaneously submitted to the same nodes without any significant loss of individual job performance; this would certainly increase resource utilization and as long as the predicted resource usage does not exceed the available resources, the time it takes to complete individual jobs should not be significantly affected. The increase of resource utilization is possible as long as the assumption that several different classes of software can be concurrently executed without significant loss of performance.

It is expected that Resource Managers could use a combination of resource selection algorithms besides the proposed DiSched component. To ensure that the available resources that the scheduler is aware of is maintained and updated, resource monitoring (i.e. Ganglia, MDS, etc…) will also be necessary. Some current

resource managers use resource monitoring to make scheduling decisions, but often only one job is normally submitted to each individual resource. However, combining resource monitoring with predictive scheduling has the potential to not only improve scheduling decisions to yield lower end-to-end job execution times, but to increase resource utilization significantly.

The proposed future work presented here is based on a technical report [92] I wrote. The report includes a more in-depth related work section, and includes an experimental results section proving some of the basic assumptions needed for the proposed future work to be realizable.

6.1 Related Work

The goals of mapping software requirements to available resources has been studied extensively, however, in practice, it is still a relatively manual process and is application specific. One approach is to perform benchmarks on resources, however this still requires user specified relationships between benchmarks and software requirements to be established. Performing these benchmarks on the resources acts as a "stepping stone" towards gaining insight about the performance of a particular set of resources, and what kind of problems could possibly benefit the most from them. Regarding application performance models, there seems to be two general methods, workflow analysis and the use of compilers to some degree. The drawbacks of workflow analysis is that it is a hard problem and time consuming to produce accurate workflows. As for compiler technology, it requires user intervention to model complex applications. Furthermore, performance models might not reflect actual performance on various different architectures.

What I propose is to seek an automatic mapping between software requirements and available resources using "black box" approach, which would be ideally generic and application independent. In principle, performance models could be built to be classified into several software classes based on the type and amount of resource usage. The models would take into consideration interactions between various components in the system, and across distributed and different systems. Best of all, applications could have performance models built without any modifications or expert knowledge about the particular software.

Some of the lessons learned from the related work are that benchmarking of Grid resources could be used to enhance the resource selection, especially in the heterogeneous systems that are often found in today's Grids. Co-scheduling could also be used based on the software classes established to increase the resource utilization. Furthermore, historical information could be used to recall the performance models generated for commonly used software in order to save the cost of generating a new model. Much of the surveyed work concentrated on lower levels of scheduling (i.e. local scheduling); it is important to address scheduling decisions on a larger global scale, and perhaps giving hints to the low level schedulers in order to improve the low level local scheduling.

6.2 Proposed Future Work

The proposed future work is aimed at developing the following three components with the goals to increasing resource utilization while decreasing end-to-end job execution times using predictive scheduling in the Grid:

- DiProfile – A Job Profiler for distributed software
- DiPredict – Performance Predictions of distributed computations
- DiSched – Resource scheduling using DiPredict in distributed environments

DiPerF is the basic building block and foundation for the DiProfile component. The DiProfile components will essentially be the DiPerF framework specialized for the task of profiling software in a fully automatic manner and outputting the needed results to the next component, namely DiPredict.

Figure 65 presents the proposed system overview. Note that the 3 components (DiProfile, DiPred, and DiSched) are placed in between the user and the Grid; the proposed system is meant to act as a run-time tool to aid in scheduling decisions. The DiProfile component breaks analyzes the software and its input and generates the needed hardware requirements and tries to classify the software class in which that particular software belongs to. The DiPredict component then attempts to use the hardware requirements and build some high level choices that the user could choose from, such as resource utilization desired, length of job desired, or maximum cost

allowed. Based on the feedback or the user and the predictions made by DiPredict, the DiSched component can then perform a matchmaking between what the user wants, with the software requirements, and the available resources.

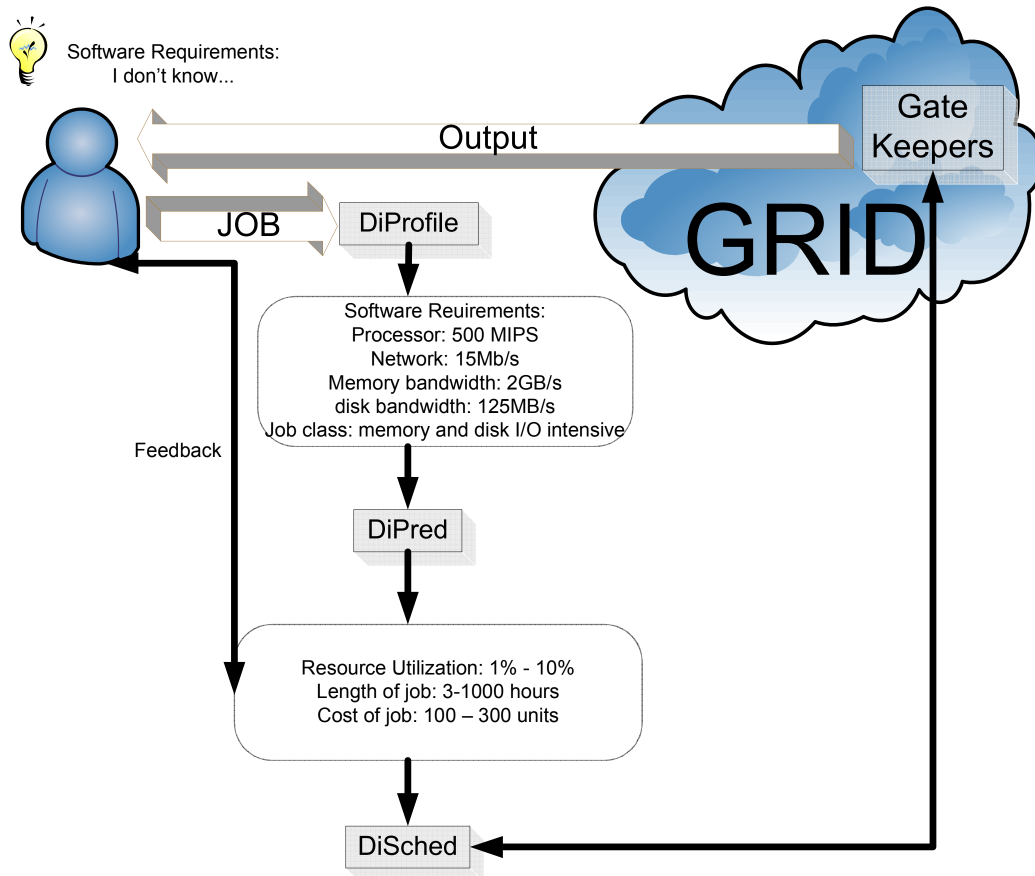


Figure 65: Proposed system overview

6.2.1 DiProfile: Perform software characterization

Modeling distributed applications (i.e. parallel computational programs) might be more challenging since the dataset on which the applications work against often influence the performance of the application, and therefore general static predictive models might not be sufficient. Using DiPerF and a small dedicated cluster of machines, we can build dynamic performance models to automatically map raw hardware resources to the performance of a particular distributed application and its representative workload; in essence, these dynamic performance models can be thought of as job profiles, and will be implemented in the component DiProfile. The intuition behind DiProfile is that based on some small sample workload (with varying sizes) and a small set of resources (with varying size), we can make predictions regarding the execution time and resource utilization of the entire job running over the complete dataset. The DiProfile stage will be a relatively expensive component in both time and computational resources, however its overhead will be warranted as long as the typical job submitted is significantly larger than the amount of time DiProfile needs to build its dynamic performance models.

We envision that several different software classes will emerge which DiProfile will be able to automatically identify which class any software belongs to; these classes would probably represent the average case resource utilization for each respective metric. Some of these classes could be:

- CPU Intensive only

- Network Intensive only
- Memory Intensive only (capacity)
- Memory Intensive only (bandwidth)
- Storage Intensive only (capacity)
- Storage Intensive only (bandwidth)
- Any combination of the above classes

The number of software classes could quickly grow due to the many combinations possible, however defining such software classes will have certain advantages especially in scalability at the sacrifice of some expressivity. One could imagine once some software is characterized and classified with a particular class, then defining rules to decide what other software could be co-scheduled concurrently could be easily realized.

The questions that remain unanswered for this section are:

- Can the performance of a complex piece of software that heavily depends on its input be characterized in its entirety in a fraction of the time that it would take to run the entire workload?
- Is the average case resource utilization sufficient to make the software class useful in real world resource management?
- How flexible will the job profiler be to different types of software classes?
- What is the maximum amount of time users are willing to wait for a job profile?
- What overall performance improvement is needed in order to offset the extra complexity, time, and resources that the software profiler introduces?

6.2.2 DiPredict: Automatic mapping of software requirements to raw resources & Service Performance Predictions

Using the data generated by DiProfile, I believe it is possible to build analytical models that estimate a service performance given some characterization of the resources that would be utilized and the current state of those resources. Previous research [96-108] has used statistical time series, regression and/or historical information in order to perform predictions in the context of service performance or network performance. On the other hand, there are other multivariate analytical models that have the potential of having better predicting accuracy. The Grid is a complex system, and often no particular metric is sufficiently comprehensive; however, an approach (such as neural networks or support vector machines) having the capability to learn relationships automatically among various dependent metrics can be a significantly more powerful approach that would yield enhanced prediction accuracy in a wider range of circumstances.

There is a gap between software requirements (high level) and hardware resources (low level). Automatic mapping could produce better scheduling decisions and give users feedback with the expected running time of their software. Using DiProfile, we can make predictions on the performance of the jobs based on the amount of raw resources dedicated to the jobs. The accuracy of the predictions will heavily rely on the idea that reliable software performance characterization is possible with only a fraction of the data input space.

The questions that remain unanswered for this section are:

- Can this automatic mapping (software requirements → hardware resources) be achieved for a wide range of software classes?
- Are the predictions accurate enough to be useful?
- How flexible/fragile are the predictions?
- Can the predictions be computed fast enough to satisfy the user agreeable wait time?

6.2.3 DiSched: Matchmaking from needed resources to available raw resources

Using DiPred, the scheduler (DiSched) can make better decisions to satisfy the requested duration of the job, where the job should be placed, etc. Since jobs are profiled based on what raw resources they will likely consume and the duration of those resource usage, multiple different jobs could be simultaneously submitted to the same nodes without any significant loss of individual job performance; this would certainly increase resource utilization and as long as the predicted resource usage does not exceed the available resources, the time it takes to complete individual jobs should not be significantly affected. The increase of resource utilization is possible as long as the assumption that several different classes of software can be concurrently executed without significant loss of performance. Figure 66 attempts to explain this very concept of co-scheduling in which 3 different applications belonging to different software classes exhibit different total running time and resource utilization depending on whether the software is run in parallel or in series.

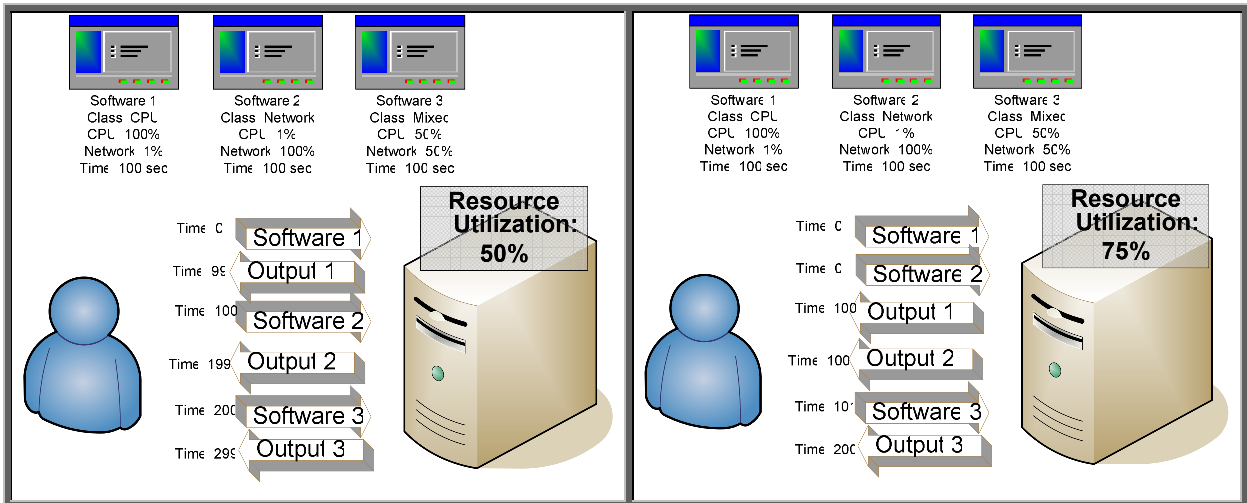


Figure 66: Simple example showing that co-scheduling software that has different requirements could be run in parallel without loss of performance; note that in a serial fashion, it would take 299 seconds to complete all 3 jobs, while if we run the first 2 jobs concurrently, it would only take 200 seconds

It is expected that Resource Managers could use a combination of resource selection algorithms besides the proposed DiSched component. To ensure that the available resources that the scheduler is aware of is maintained and updated, resource monitoring (i.e. Ganglia, MDS, etc…) will also be necessary. Some current resource managers use resource monitoring to make scheduling decisions, but often only one job is normally submitted to each individual resource. However, combining resource monitoring with predictive scheduling has the potential to not only improve scheduling decisions to yield lower end-to-end job execution times, but to increase resource utilization significantly.

Using DiSched, resource managers / brokers can make more informed decisions regarding resource selection, resource allocation / reservation, load balancing, and satisfying a particular Quality of Service (QoS). There has been much work (Condor, PBS, LSF, MAUI, in clusters and CSF in Grids [109-116]) in the area of resource managers and resource brokers that use the state of current resources to make scheduling decisions. The limitation of this approach is that heuristics must be found that finds relationships between the state of current resources and the application requirements, something that is currently only automatically realized at a basic level. Furthermore, the state of the available resources could both be incomplete and/or stale, and therefore decisions made solely on this state could be suboptimal. Finally, collecting the state of the available resources and always having up-to-date information is not a scalable approach. The *overall objective* is to develop a scalable resource selection algorithm that utilizes both the state of the current resources and the predictive models that contain dynamic run-time information. The *central hypothesis* is that current resource allocation algorithms could be improved and as the Grid grows, the problems with the current resource allocation techniques will become even more apparent as the state of the available resources becomes harder to keep up to date. The *hypothesis will be tested* by the comparison between the proposed resource allocation algorithm and existing approaches; the performance characteristics will be the resource utilization, the flexibility of the

approach under varying conditions, and the scalability of the compared approaches. The *expected outcome* is a scalable resource scheduler that uses both predictive models and available resource information to make better resource scheduling decisions that yield higher resource utilization and that scales along with the growing Grid. My *long term career goal* is to integrate the proposed resource allocation algorithm into existing resource managers and brokers. Ensuring that the performance of resource managers will be maintained as the size of the systems they coordinate grows will be essential to the scalability of the Grid. The *expected significance* consists of the automatic mapping between application requirements and raw resources which would lead to better resource selection algorithms; this essentially would allow higher resource utilization in the Grid, and hence make the Grid more scalable.

The questions that remain unanswered for this section are:

- Can software performance predictions aid resource scheduling decisions in a consistent and significant manner?
- Can a broad type of software (that identifies not resource usage, but rather software features, workload, usage patterns, etc) be defined that will most likely benefit from DiSched?
- Are dynamic run-time performance models more accurate than static generic performance models, especially for certain types of software?

6.3 Future Work Conclusion

The proposed future work has been addressed in further detail in a technical report [92]. Between that report and this future work description, I have covered the following items:

- A literature survey of resource management using predictive scheduling
- Showed that co-scheduling is possible for several different classes of software without significant loss of performance
- Showed that reliable software performance characterization is possible with only a fraction of the entire input space
- A proposal outlining the entire system of components and the related work
- Showed that reliable software performance characterization is possible with only a fraction of the entire input space (at least for 2 specific problems – Jacobi 2D and quick sort)

The proposed future work is aimed at developing the following three components with the goals to increasing resource utilization while decreasing end-to-end job execution times using predictive scheduling in the Grid:

- DiProfile – A Job Profiler for distributed computations
- DiPredict – Performance Predictions of distributed computations
- DiSched – Resource scheduling using DiPredict in distributed environments

I believe that the proposed research addresses significant work in the area of performance modeling and resource management. My research work should play a critical role in understanding important problems in the design, development, management and planning of complex and dynamic systems and could enhance the scalability of the Grid as well as improve resources utilization despite the Grid's growing size.

7 Conclusion

As presented in this work, performing distributed measurements is not a trivial task, due to difficulties 1) *accuracy* – synchronizing the time across an entire system that might have large communication latencies, 2) *flexibility* – in heterogeneity normally found in WAN environments and the need to access large number of resources, 3) *scalability* – the coordination of large amounts of resources, and 4) *performance* – the need to process large number of transactions per second. In attempting to address these four issues, we developed DiPerF, a DIstributed PERformance testing Framework, aimed at simplifying and automating service performance evaluation. DiPerF coordinates a pool of machines that test a single or distributed target service (i.e. grid services, network services, distributed services, etc), collects and aggregates performance metrics (i.e. throughput, service response time, etc) from the client point of view, and generates performance statistics (fairness of resource utilization, saturation point of service, scalability of service, etc). The aggregate data collected provides information on service throughput, service response time, on service 'fairness' when serving multiple clients concurrently, and on the impact of network latency on service performance. Furthermore, using the collected data, it is possible to build predictive models that estimate service performance as a function of service load. Using the power of this framework, we have analyzed the performance scalability of the several components of the Globus Toolkit and several grid services. We measured the performance of various components of the GT in a wide area network (WAN) as well as a local area network (LAN) with the goal of understanding the performance that is to be expected from the GT in a realistic deployment in a distributed and heterogeneous environment.

The contributions of this thesis are two fold: 1) a detailed empirical performance analysis of various components of the Globus Toolkit along with a several grid services, and 2) DiPerF, a tool that makes automated distributed performance testing easy.

Through our tests performed on GRAM, WS-MDS, and GridFTP, we have been able to quantify the performance gain or loss between various different versions or implementations, and have normally found the upper limit on both scalability and performance on these services. We have also been able to show the performance of these components in a WAN, a task that would have been very tedious and time consuming without a tool such as DiPerF. By pushing the Globus Toolkit to the limit in both performance and scalability, we were able to give the users a rough overview of the performance they are to expect so they can do better resource planning. The developers also gained feedback on the behavior of the various components under heavy stress and allowed them to concentrate on improving the parts that needed the most improvements. We were also able to quantify the performance and scalability of DI-GRUBER, a distributed grid service built on top of the Globus Toolkit 3.2 and the Globus Toolkit 3.9.5.

The second main contribution is DiPerF itself, which is a tool that allows large scale testing of grid services, web services, network services, and distributed services to be done in both LAN and WAN environments. DiPerF has been automated to the extent that once configured, the framework will automatically do the following steps:

- check what machines or resources are available for testing
- deploy the client code on the available machines
- perform time synchronization
- run the client code in a controlled and predetermined fashion
- collect performance metrics from all the clients
- stop and clean up the client code from the remote resources
- aggregate the performance metrics at a central location
- summarize the results
- generates graphs depicting the aggregate performance of the clients and tested service

Some lessons we learned through the work presented in this thesis are:

- building scalable software is not a trivial task
    - there were some interesting issues we encountered when we tried to scale DiPerF beyond a few hundred clients, but after careful tuning and optimizations, we were able to scale DiPerF to 10,000+ clients
    - ssh is quite heavy weight for a communication channel, so for a real scalable solution, proprietary TCP/IP or UDP/IP communication channels are recommended
- time synchronization is a big issue when doing distributed measurements in which the aggregate view is important; although NTP offers potentially accurate clock synchronization, due to mis-configurations and possibly not wide enough deployment, NTP is not sufficient in a general case
- C-based components of the Globus Toolkit normally perform significantly better than their Java counterparts
- depending on the particular service tested, WAN performance is not always comparable to that found in a LAN; for example, WS-MDS with no security performed comparable between LAN and WAN tests, but WS-MDS with security enabled achieved less than half the throughput in a WAN when compared to the same test in a LAN
- the testbed performance (in our case it was mostly PlanetLab) can influence the performance results, and hence careful care must be taken in comparing experiments done at different times when the state of PlanetLab could have significantly changed

We conclude with the thought that we succeeded in building a scalable and high performance measurements tool that can be used to coordinate, measure, and aggregate the performance of thousands of clients distributed all over the world targeting anything from network services, web services, distributed services, to grid services. We have shown DiPerF's accuracy as being very good with only a few percent of performance deviation between the aggregate client view and the centralized service view. We have also contributed towards a better understanding of various vital Globus Toolkit components such as GRAM, MDS, and GridFTP.

8 Bibliography

[1] The Globus Alliance, www.globus.org.

[2] I. Foster, C. Kesselman "The Grid 2: Blueprint for a New Computing Infrastructure", "Chapter 1: Perspectives." Elsevier Publisher, 2003.

[3] I. Foster, C. Kesselman, S. Tuecke, "The Anatomy of the Grid", International Supercomputing Applications, 2001.

[4] I. Foster, C. Kesselman, J. Nick, S. Tuecke. The Physiology of the Grid: An Open Grid Services Architecture for Distributed Systems Integration. Open Grid Service Infrastructure WG, Global Grid Forum, June 22, 2002.

[5] The Globus Alliance, "WS GRAM: Developer's Guide", http://www-unix.globus.org/toolkit/docs/3.2/gram/ws.

[6] L. Peterson, T. Anderson, D. Culler, and T. Roscoe, "A Blueprint for Introducing Disruptive Technology into the Internet", Proceedings of the First ACM Workshop on Hot Topics in Networking (HotNets), October 2002.

[7] A. Bavier et al., "Operating System Support for Planetary-Scale Services", Proceedings of the First Symposium on Network Systems Design and Implementation (NSDI), March 2004.

[8] I. Foster, et al., "The Grid2003 Production Grid: Principles and Practice", 13th IEEE Intl. Symposium on High Performance Distributed Computing, 2004.

[9] D. Gunter, B. Tierney, C. E. Tull, V. Virmani, On-Demand Grid Application Tuning and Debugging with the NetLogger Activation Service, 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 17th, 2003.

[10] G. Tsouloupas, M. Dikaiakos. "GridBench: A Tool for Benchmarking Grids," 4th International Workshop on Grid Computing, Grid2003, Phoenix, Arizona, November 17th, 2003.

[11] The Globus Alliance, "Globus Toolkit 3.0 Test Results Page", http://www-unix.globus.org/ogsa/tests/gt3_tests_result.html

[12] The Globus Alliance, "Overview and Status of Current GT Performance Studies", http://www-unix.globus.org/toolkit/docs/development/3.9.5/perf_overview.html

[13] W. Allcock, J. Bester, J. Bresnahan, I. Foster, J. Gawor, J. A. Insley, J. M. Link, and M. E. Papka. "GridMapper, A Tool for Visualizing the Behavior of Large-Scale Distributed Systems," 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11), pp179-187, Edinburgh, Scotland, July 24-16, 2002.

[14] C. Lee, R. Wolski, I. Foster, C. Kesselman, J. Stepanek. "A Network Performance Tool for Grid Environments," Supercomputing '99, 1999.

[15] R. Wolski, N. Spring, J. Hayes, "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing," Future Generation Computing Systems, 1999.

[16] The Globus Alliance, "GT3 GRAM Tests Pages", http://www-unix.globus.org/ogsa/tests/gram.

[17] X. Zhang, J. Freschl, and J. Schopf. "A Performance Study of Monitoring and Information Services for Distributed Systems." Proceedings of HPDC, August 2003.

[18] X. Zhang and J. Schopf. "Performance Analysis of the Globus Toolkit Monitoring and Discovery Service, MDS2." Proceedings of the International Workshop on Middleware Performance (MP 2004), April 2004.

[19] G. Aloisio, M. Cafaro, I. Epicoco, and S. Fiore, “Analysis of the Globus Toolkit Grid Information Service”. Technical report GridLab-10-D.1-0001-GIS_Analysis, GridLab project, http://www.gridlab.org/Resources/Deliverables/D10.1.pdf.

[20] B. Allcock, J. Bresnahan, R. Kettimuthu, M. Link, C. Dumitrescu, I. Raicu, I. Foster. "Zebra: The Globus Striped GridFTP Framework and Server", submitted for review.

[21] G. Kola, T. Kosar, and M. Livny. “Profiling Grid Data Transfer Protocols and Servers”, Proceedings of Euro-Par 2004, September 2004.

[22] T. Baer, P. Wyckoff. “A Parallel I/O Mechanism for Distributed Systems”, Proceedings of Cluster '04, San Diego, CA, September 2004.

[23] X. Liu, H. Xia, and A.A. Chien, "Network Emulation Tools for Modeling Grid Behavior," 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003), May 12-15, 2003 in Tokyo, Japan.

[24] C. Dumitrescu, I. Foster. "GRUBER: A Grid Resource SLA-based Broker", EuroPar 2005.

[25] C. Dumitrescu, I. Raicu, I. Foster. "DI-GRUBER: A Distributed Approach for Grid Resource Brokering", submitted for review to SC 2005.

[26] C. Dumitrescu, I. Foster, I. Raicu. "A Scalability and Performance Evaluation of a distributed Usage SLA-based Broker in Large Grid Environments", GriPhyN/iVDGL Technical Report, March 2005.

[27] C. Dumitrescu, I. Raicu, I. Foster. “Performance Measurements in Running Workloads over a Grid”, submitted for review to SC 2005.

[28] A. Adams, J. Mahdavi, M. Mathis, and V. Paxson. Creating a scalable architecture for Internet measurement. IEEE Network, 1998.

[29] V. Paxson, J. Mahdavi, A. Adams, and M. Mathis. An architecture for large-scale internet measurement. IEEE Communications, 36(8):48–54, August 1998.

[30] V. Paxson. End-to-end Internet packet dynamics. In Proceedings of ACM SIGCOMM ’97, Cannes, France, September 1997.

[31] N. Anerousis, R. C´aceres, N. Duffield, A. Feldmann, A. Greenberg, C. Kalmanek, P. Mishra, K. Ramakrishnan, and J. Rexford. Using the AT&T labs PacketScope for Internet measurement. AT&T Services and Infrastructure Performance Symposium, November 1997.

[32] A. Feldmann. Continuous on-line extraction of HTTP traces from packet traces. Position paper: W3C Web Characterization Workshop, November 1998.

[33] Internet Protocol Performance Metrics. http://www.advanced.org/ippm/index.html, 1998.

[34] V. Paxson, G. Almes, J. Mahdavi, and M. Mathis. Framework for IP performance metrics. IETF RFC 2330, 1998.

[35] The Surveyor Project. http://www.advanced.org/csgippm/, 1998.

[36] National Laboratory for Applied Network Research. http://www.nlanr.net, 1998.

[37] NLANR Acitve Measurement Program - AMP. http://moat.nlanr.net/AMP.

[38] NLANR Passive Measurement and Analysis - PMA. http://moat.nlanr.net/PMA.

[39] A Distributed Testbed for National Information Provisioning. http://ircache.nlanr.net/Cache/.

[40] Keynote Systems Inc. http://www.keynote.com, 1998.

[41] Collaborative Advanced Interagency Research Network. http://www.cairn.net.

[42] Cooperative Association for Internet Data Analysis. http://www.caida.org, 1998.

[43] CAIDA. “Cooperative Association for Internet Data Analysis.” CAIDA, 1996.

[44] Skitter. http://www.caida.org/tools/measurement/skitter.

[45] V. Jacobson. traceroute. ftp://ftp.ee.lbl.gov/traceroute.tar.Z, 1989.

[46] Coral: Passive network traffic monitoring and statistics collection. http://www.caida.org/tools/coral.

[47] C.R. Simpson Jr., G.F. Riley: NETI@home: A Distributed Approach to Collecting End-to-End Network Performance Measurements. PAM 2004: 168-174

[48] Ch. Steigner and J. Wilke, "Isolating Performance Bottlenecks in Network Applications", in Proceedings of the International IPSI-2003 Conference, Sveti Stefan, Montenegro, October 4-11, 2003.

[49] B.B. Lowekamp, N. Miller, R. Karrer, T. Gross, and P. Steenkiste, "Design, Implementation, and Evaluation of the Remos Network Monitoring System," Journal of Grid Computing 1(1):75--93, 2003.

[50] IEPM. Internet End-to-End and Process Monitoring (SLAC/DOE). http://wwwiepm.slac.stanford.edu/

[51] MAWI (WIDE Project), http://www.wide.ad.jp/wg/mawi/

[52] PPNCG Network Monitoring, http://icfamon.rl.ac.uk/ppncg/main.html

[53] TRIUMF Network Monitoring, http://sitka.triumf.ca/

[54] WAND (Waikato Applied Network Dynamics) WITS (Waikato Internet Traffic Storage) Project, http://wand.cs.waikato.ac.nz/

[55] Andover News Network's Internet Traffic Report, http://www.internettrafficreport.com/main.htm

[56] MIDS Internet Average, http://**average**.miq.net/

[57] MIDS Internet Weather Report, http://www.mids.org/weather/

[58] MIDS Matrix IQ Ratings Comparing Performance of Some ISPs, http://ratings.miq.net/

[59] NetSizer (Telcordia Technologies), http://www.netsizer.com/

[60] P. Barford ME Crovella. Measuring Web performance in the wide area. Performance Evaluation Review, Special Issue on Network Traffic Measurement and Workload Characterization, August 1999.

[61] G. Banga and P. Druschel. Measuring the capacity of a Web server under realistic loads. World Wide Web Journal (Special Issue on World Wide Web Characterization and Performance Evaluation), 1999.

[62] I. Foster and C. Kesselman, Eds., "The Grid: Blueprint for a Future Computing Infrastructure", "Chapter 2: Computational Grids." Morgan Kaufmann Publishers, 1999.

[63] Grid3. http://www.ivdgl.org/grid3/

[64] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An Overlay Testbed for Broad-Coverage Services," ACM Computer Communications Review, vol. 33, no. 3, July 2003.

[65] Information Sciences Institute, University of Southern California, "Internet Protocol," Request for Comments 791, Internet Engineering Task Force, September 1981.

[66] Defense Advanced Research Projects Agency & Information Sciences Institute, University of Southern California, "Transmission Control Protocol," Request for Comments 793, Internet Engineering Task Force, September 1981.

[67] J. Postel, ISI, "User Datagram Protocol," Request for Comments 768, Internet Engineering Task Force, August 1980.

[68] T. Ylonen, T. Kivinen, M. Saarinen, T. Rinne, S. Lehtinen, "SSH Protocol Architecture," Network Working Group, Internet-Draft, November 19, 2001.

[69] S. Bradner, A. Mankin, "IP: Next Generation (IPng) White Paper Solicitation," Request for Comments 1550, Internet Engineering Task Force, December 1993

[70] S. Deering, R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification," Request for Comments 1883, Internet Engineering Task Force, December 1995.

[71] Tuecke, S., Czajkowski, K., Foster, I., Frey, J., Graham, S., Kesselman, C, Maguire T., Sandholm, T., Snelling, D., and Vanderbilt, P., Open Grid Services Infrastructure (OGSI) Version 1.0. Global Grid Forum, June 2003.

[72] "GT3 Core Key Concepts". http://www-unix.globus.org/toolkit/docs/3.2/core/key/index.html

[73] "GT4 Release Contents". http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/GT4Facts/index.html#Contents

[74] "GRAM: Key Concepts". http://www-unix.globus.org/toolkit/docs/3.2/gram/key/index.html

[75] K. Czajkowski, I. Foster, N. Karonis, C. Kesselman, S. Martin, W. Smith, S. Tuecke, "A Resource Management Architecture for Metacomputing Systems", IPPS/SPDP '98 Workshop on Job Scheduling Strategies for Parallel Processing, pg. 62-82, 1998.

[76] "GT 4.0 WS GRAM Approach". http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/execution/key/WS_GRAM_Approach.html

[77] GridFTP: Universal Data Transfer for the Grid. Globus Project, White Paper. http://www.globus.org/datagrid/deliverables/C2WPdraft3.pdf

[78] "GT 4.0 Component Fact Sheet: WS MDS (MDS4)". http://www-unix.globus.org/toolkit/docs/development/4.0-drafts/info/WSMDSFacts.html

[79] K. Ranganathan, I. Foster, "Simulation Studies of Computation and Data Scheduling Algorithms for Data Grids", Journal of Grid Computing, 2003, 1 (1).

[80] K. Ranganathan, I. Foster, "Decoupling Computation and Data Scheduling in Distributed Data-Intensive Applications", in 11th IEEE International Symposium on High Performance Distributed Computing. 2002. Edinburgh, Scotland: IEEE Computer Society Press.

[81] C. Dumitrescu, I. Foster, "Usage Policy-based CPU Sharing in Virtual Organizations", in 5th International Workshop in Grid Computing, 2004, Pittsburg, PA.

[82] C. Dumitrescu, I. Raicu, M. Ripeanu, I. Foster. "DiPerF: an automated DIstributed PERformance testing Framework." 5th International IEEE/ACM Workshop in Grid Computing, 2004, Pittsburg, PA.

[83] Bill Wilson. "GKrellM Monitor," http://members.dslextreme.com/users/billw/gkrellm/gkrellm.html

[84] "select(2) - Linux man page", http://www.die.net/doc/linux/man/man2/select.2.html

[85] Abhishek Chandra and David Mosberger. "Scalability of Linux Event-Dispatch Mechanisms," Proceedings of the USENIX Annual Technical Conference (USENIX 2001), Boston, MA, June 2001.

[86] N. Minar, "A Survey of the NTP protocol", MIT Media Lab, Dcemeber 1999, http://xenia.media.mit.edu/~nelson/research/ntp-survey99.

[87] T. Williams, C. Kelley. "gnuplot, An Interactive Plotting Program", http://www.gnuplot.info/docs/gnuplot.pdf

[88] The Ganglia Distributed Monitoring System: Design, Implementation, and Experience. Matthew L. Massie, Brent N. Chun, and David E. Culler. Parallel Computing, Vol. 30, Issue 7, July 2004.

[89] C. Dumitrescu, I. Foster, "GangSim: A Simulator for Grid Scheduling Studies", Cluster Computing and Grid (CCGrid), Cardiff, UK, May 2005.

[90] C. Dumitrescu, I. Foster, "GRUBER: A Grid Resource SLA Broker", GriPhyN/iVDGL Technical Report, 2005.

[91] C. Dumitrescu, I. Raicu, M. Ripeanu. "Extending a distributed usage SLA resource broker with overlay networks to support Large Dynamic Grid Environments", work in progress.

[92] I. Raicu. "Decreasing End-to-End Job Execution Times by Increasing Resource Utilization using Predictive Scheduling in the Grid", Technical Report, Grid Computing Seminar, Department of Computer Science, University of Chicago, March 2005.

[93] H. Subramanian. "Systems Performance Evaluation Methods for Distributed Systems Using Datastreams", Master Thesis, Information & Telecommunication Technology Center, University of Kansas, January 2005.

[94] C. Pistol, A. Lungu. "Deploying C++ Grid Services: Options and Performance", Technical Report, Federated Distributed Systems, Department of Computer Science, Duke University, December 2004.

[95] A. Dan, C. Dumitrescu, and M. Ripeanu, "Connecting Client Objectives with Resource Capabilities: An Essential Component for Grid Service Management Infrastructures," 2nd International Conference on Service Oriented Computing (ICSOC), November 2004, New York, NY.

[96] R. Wolski, "Dynamically Forecasting Network Performance Using the Network Weather Service", Journal of Cluster Computing, Volume 1, pp. 119-132, Jan. 1998.

[97] A. Danalis, C. Dovrolis. "ANEMOS, An Autonomous NEtwork MOnitoring System." 4th Passive and Active Measurements (PAM) Workshop 2003.

[98] U. Hofmann, I. Milouchewa1. "Distributed Measurement and Monitoring in IP Networks." SCI 2001/ISAS 2001 Orlando 7/2001.

[99] J. L. Hellerstein, M. M. Maccabee, W. Nathaniel Mills III, and J. J. Turek. "ETE, A Customizable Approach to Measuring End-to-End Response Times Their Components in Distributed Systems." 19th IEEE International Conference on Distributed Computing Systems (ICDCS) 1999.

[100] C. R. Simpson Jr., G. F. Riley: "NETI@home: A Distributed Approach to Collecting End-to-End Network Performance Measurements." PAM 2004.

[101] J. Schopf and F. Berman, "Using Stochastic Information to Predict Application Behavior on Contended Resources," Jrnl. of Foundations of CS, 2001.

[102] W. Smith, I. Foster, and V. Taylor. "Predicting Application Run Times Using Historical Information". IPPS/SPDP 1998.

[103] P. Dinda, "A Prediction-based Real-time Scheduling Advisor", 16th International Parallel and Distributed Processing Symposium (IPDPS 2002).

[104] D.A. Bacigalupo, S.A. Jarvis, L. He, D.P. Spooner, D.N. Dillenberger, G.R. Nudd, "An investigation into the application of different performance prediction techniques to distributed enterprise applications", Journal of Supercomputing, 2004.

[105] S.A. Jarvis, D.P. Spooner, H.N. Lim Choi Keung, J. Cao, S. Saini, G.R. Nudd. "Performance Prediction and its use in Parallel and Distributed Computing Systems", IEEE/ACM International Workshop on Performance Modelling, Evaluation and Optimization of Parallel and Distributed Systems, 2003.

[106] F. Vraalsen. "Performance Contracts: Predicting and Monitoring Grid Application Behavior." MS Thesis, Graduate College of UIUC, 2001.

[107] N. N. Tran. "Automatic ARIMA Time Series Modeling and Forecasting for Adaptive Input/Output Prefetching." PhD thesis, UIUC, 2001.

[108] J. Oly and D. A. Reed, "Markov Model Prediction of I/O Request for Scientific Application," FAST 2002.

[109] D. Irwin, J. Chase, and L. Grit, "Balancing Risk and Reward in Market-Based Task Scheduling." HPDC-13, 2004.

[110] J. Kay, and P. Lauder, "A Fair Share Scheduler," University of Sydney and AT&T Bell Labs, 1988.

[111] "Maui Scheduler, http://www.supercluster.org/maui/." Center for HPC Cluster Resource Management and Scheduling, 2004.

[112] LSF Administrator's Guide, Version 4.1, Platform Computing Corporation, February 2001.

[113] Condor Project, A Resource Manager for High Throughput Computing, Software Project, The University of Wisconsin, www.cs.wisc.edu/condor.

[114] OpenPBS Project, A Batching Queuing System, Software Project, Altair Grid Technologies, LLC, www.openpbs.org.

[115] I. Foster, C. Kesselman, C. Lee, R. Lindell, K. Nahrstedt, and A. Roy, "A Distributed Resource Management Architecture that Supports Advance Reservations and Co-Allocation," Proc. International Workshop on Quality of Service, 1999.

[116] "Open source metascheduling for Virtual Organizations with the Community Scheduler Framework (CSF)", Technical White Paper, Platform, 2003.